DISTRIBUTED EXECUTION OF RECURSIVE IRREGULAR APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Nikhil D. Hegde

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Milind Kulkarni

School of Electrical and Computer Engineering

Dr. Samuel P. Midkiff

School of Electrical and Computer Engineering

Dr. Y. Charlie Hu

School of Electrical and Computer Engineering

Dr. Anand Raghunathan

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

Head of the School Graduate Program

Dedicated to my parents.

ACKNOWLEDGMENTS

My sincere thanks to Prof. Milind Kulkarni for his time and effort in mentoring me throughout my Ph.D. studies. Milind has inspired me to do research, teach, and is a role model for a mentor. My heartfelt gratitude towards him. I am fortunate to have great advisory committee members, who provided valuable feedback that ensured my progress. Many thanks to Prof. Samuel Midkiff for generously providing infrastructure to conduct my experiments, Prof. Anand Raghunathan for motivating me to join the Ph.D. program at Purdue, and Prof. Charlie Hu for asking those tough questions to help improve my work. I also owe thanks to anonymous reviewers for providing valuable feedback on my research, to Joanne Lax for the helpful professional development workshops. Special thanks to Prof. V. Kamakoti and Prof. C. Siva Ram Murthy from IIT Madras for mentoring and eagerly supporting my efforts to join Ph.D. studies.

I am thankful to have a supporting family including my brother, in-laws, especially my wife Bhavya, without whose support this journey would not have been possible. My kids Mayank and Maanya motivate and inspire me to learn everyday. I would also like to thank my lab colleagues Jianqiao, Kirshanthan, Jad, Nour, Laith, and Chris for many interesting discussions. Lastly but not the least, I would like to thank the staff in my department and the wonderful people of West Lafayette/Lafayette area for their very friendly and welcoming behavior.

The research presented in this thesis was made possible through grants NSF CCF-1150013, DOE DE-SC0010295, and XSEDE TG-ASC170007.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

ABSTRACT xi

1 INTRODUCTION 1

1.1 Contributions 6

1.2 Outline 7

2 SPIRIT: A FRAMEWORK FOR CREATING DISTRIBUTED RECURSIVE TREE APPLICATIONS 8

2.1 Introduction 8

2.2 Background and Motivation 10

2.2.1 Terminology and Traversal Characteristics 11

2.2.2 Tree Traversal Algorithms and Locality 12

2.2.3 Graph Processing Frameworks 13

2.3 Design 15

2.3.1 Tree Partitioning and Distribution 15

2.3.2 Tree Traversals and Pipeline Parallelism 18

2.3.3 Scheduling for Locality and Aggregation 19

2.3.4 Load Balance and Space Adaptivity 21

2.4 Implementation 22

2.5 Evaluation 24

2.5.1 Methodology 25

2.5.2 Scalability 27

2.5.3 Space-adaptive Evaluation 30

2.5.4 Performance Breakdown 33

Page

2.5.5 SPIRIT, DGL, PBGL, and Reference Software 35
2.6 Related Work 37
2.7 Conclusions 39
3 TREELOGY: A BENCHMARK SUITE FOR TREE TRAVERSALS 40
3.1 Introduction 40
3.1.1 Contributions 41
3.1.2 Outline 42
3.2 Background 43
3.2.1 Trees for Accelerating Computations 43
3.2.2 Traversal Structure and Optimizations 47
3.3 Treelogy Traversal Kernels 48
3.4 An Ontology for Tree Traversals 51
3.4.1 Ontology 51
3.4.2 Optimizations 53
3.5 Evaluation 56
3.5.1 Methodology 56
3.5.2 Scalability 61
3.5.3 Case Studies 64
3.6 Related Work 66
3.7 Conclusions 67
4 D2P: FROM RECURSIVE FORMULATIONS TO DISTRIBUTED-MEMORY CODES 68
4.1 Introduction 68
4.1.1 Overview 70
4.2 Background and Motivation 74
4.3 Design 76
4.3.1 Specification 76
4.3.2 Inspector and Executor 78

Page

4.3.3 Design Details 90
4.4 Implementation 94
4.5 Evaluation 98
4.5.1 Methodology 99
4.5.2 Scalability 102
4.5.3 Case Studies 107
4.6 Related work 109
4.7 Conclusions 111
5 CONCLUSIONS 113
REFERENCES 115
VITA 124

LIST OF TABLES

Table Page

2.1 Data sets; |V|= Vertex set size, |P|=Number of points. 26

2.2 Process utilization: S=Dataset Size, P=Number of processes, TT=Total traversal time (seconds), VT=per-process computation time as a percentage of total traversal time, D=aggregate volume of data exchanged in MBytes. 32

3.1 Classification of benchmarks 54

3.2 Data sets and attributes; |V|= Number of vertices, |P|=Number of traversals. 58

3.3 Runtime characteristics of benchmarks. 59

4.1 Auxiliary methods in D2P. 97

4.2 Number of tasks and the unroll depth used in strong-scaling experiments. The unroll depth used for MCM is the same as that of MWT and hence, is not shown. 100

4.3 Details of (i) baseline (1_rec) and preprocessing overhead (Pre) runtimes in seconds and (ii) Comparison of speedups ($\times$) obtained with 1 and W Cilk workers per process over baseline runtimes. The single Cilk worker per process numbers are from Figure 4.5. W represents the workers corresponding to the best run obtained from a sweep of 1 to 16 Cilk workers. 103

LIST OF FIGURES

Figure Page

2.1 Tree Traversal in Nearest Neighbor. 11

2.2 Tree partitioning and distribution in SPIRIT. 15

2.3 Distributed octree building. 17

2.4 Block traversal through distributed tree. 20

2.5 SPIRIT Architecture, API, and example program 23

2.6 Strong scaling in SPIRIT. 28

2.7 Weak scaling in SPIRIT. 30

2.8 Space adaptivity in SPIRIT with varying amount of replication 31

2.9 Load distribution among 128 processes. 33

2.10 Impact of a) Subtree height. b) Aggregation. 33

2.11 SPIRIT vs. ChaNGa. 35

3.1 Treelogy use case. 42

3.2 Sample space (2D) and corresponding kd-tree. 44

3.3 Purpose of spatial acceleration structures and example traversal codes . . . 45

3.4 Scaling in Treelogy benchmarks. x-axis=Thread count (SHM)/Process count (DM), y-axis (log scale) =Runtime (s). 62

3.5 GPU scalability 64

3.6 Case studies: a) estimating locality benefits using reuse distance. b) improving locality through reordered traversal schedule. c) improving load-balance through subtree replication. 65

4.1 D2P system overview. 70

Figure Page

4.2 The Minimum Weight Triangulation problem: the recurrence equation computes the least cost of triangulating a convex polygon. This equation can be thought of as computing the cells of an upper triangular matrix. The figure shows the standard implementation scheme of computing the cells using an iterative code. Also shown is the order in which the code computes those cells. 73

4.3 The specification: a) showing the outline of a recursive algorithm for the MWT problem. (b) shows the operation of this recursive formulation. . . . 76

4.4 D2P inspector in action: (a) shows task creation by unrolling MWT's top-level recursive method A two levels deep. The leaves of the recursion tree are identified as tasks. The Figure also shows the numbers of tiles read and written by each task (numbering follows from Figure 4.2). (b) shows inter-task dependences. (c) shows task partitioning among 4 processes. 91

4.5 Strong scaling in D2P benchmarks showing speedup of single Cilk worker/process configuration over baselines. SW and RNA use baseline runtimes normalized w.r.t. 2-process (2_DM) run instead of using the 1_rec baseline runtimes. Also shown is a comparison of speedups obtained with the default unrolling depth, and the best unrolling depth empirically measured. 101

4.6 Summarizing Table 4.3 results. Exploiting intra-task parallelism is necessary in compute-bound benchmarks. 104

4.7 Weak scaling in D2P benchmarks. Y-axis shows the normalized runtime w.r.t. the 8-process run. The data-labels show the input size used for that execution. 105

4.8 Case studies comparing D2P with other systems. 107

ABSTRACT

Hegde, Nikhil D. Ph.D., Purdue University, August 2019. Distributed Execution of Recursive Irregular Applications. Major Professor: Milind Kulkarni.

Massive computing power and applications running on this power, primarily confined to expensive supercomputers a decade ago, have now become mainstream through the availability of clusters with commodity computers and high-speed interconnects running big-data era applications. The challenges associated with programming such systems, for effectively utilizing the computing power, have led to the creation of intuitive abstractions and implementations targeting average users, domain experts, and savvy (parallel) programmers. There is often a trade-off between the ease of programming and performance when using these abstractions. This thesis develops tools to bridge the gap between ease of programming and performance of irregular programs—programs that involve one or more of irregular- data structures, control structures, and communication patterns—on distributed-memory systems.

Irregular programs are focused heavily in domains ranging from data mining to bioinformatics to scientific computing. In contrast to regular applications such as stencil codes and dense matrix-matrix multiplications, which have a predictable pattern of data access and control flow, typical irregular applications operate over graphs, trees, and sparse matrices and involve input-dependent data access pattern and control flow. This makes it difficult to apply optimizations such as those targeting locality and parallelism to programs implementing irregular applications. Moreover, irregular programs are often used with large data sets that prohibit single-node execution due to memory limitations on the node. Hence, distributed solutions are necessary in order to process all the data.

In this thesis, we introduce SPIRIT, a framework consisting of an abstraction and a space-adaptive runtime system for simplifying the creation of distributed implementations of recursive irregular programs based on spatial acceleration structures. SPIRIT addresses the insufficiency of traditional data-parallel approaches and existing systems in effectively parallelizing computations involving repeated tree traversals. SPIRIT employs locality optimizations applied in a shared-memory context, introduces a novel pipeline-parallel approach to execute distributed traversals, and trades-off performance with memory usage to create a space-adaptive system that achieves a scalable performance, and outperforms implementations done in contemporary distributed graph processing frameworks.

We next introduce Treelogy to understand the connection between optimizations and tree-algorithms. Treelogy provides an ontology and a benchmark suite of a broader class of tree algorithms to help answer: (i) is there any existing optimization that is applicable or effective for a new tree algorithm? (ii) can a new optimization developed for a tree algorithm be applied to existing tree algorithms from other domains? We show that a categorization (ontology) based on structural properties of tree-algorithms is useful for both developers of new optimizations and new tree algorithm creators. With the help of a suite of tree traversal kernels spanning the ontology, we show that GPU, shared-, and distributed-memory implementations are scalable and the two-point correlation algorithm with vptree performs better than the standard kdtree implementation.

In the final part of the thesis, we explore the possibility of automatically generating efficient distributed-memory implementations of irregular programs. As manually creating distributed-memory implementations is challenging due to the explicit need for managing tasks, parallelism, communication, and load-balancing, we introduce a framework, D2P, to automatically generate efficient distributed implementations of recursive divide-conquer algorithms. D2P automatically generates a distributed implementation of a recursive divide-conquer algorithm from its specification, which is a high-level outline of a recursive formulation. We evaluate D2P with recursive

Dynamic programming (DP) algorithms. The computation in DP algorithms is not irregular *per se*. However, when distributed, the computation in efficient recursive formulations of DP algorithms requires irregular communication. User-configurable knobs in D2P allow for tuning the amount of available parallelism. Results show that D2P programs scale well, are significantly better than those produced using a state-of-the-art framework for parallelizing iterative DP algorithms, and outperform even hand-written distributed-memory implementations in most cases.

1. INTRODUCTION

Big-data era has ushered in exciting applications requiring massive computing horsepower to run well. Thanks to application-driven demand and technological evolution, systems delivering such computing power are now mainstream. What was once the realm of national labs housing supercomputers running scientific applications is now easily accessible through clusters of commodity computers having high-speed interconnects and running emerging applications. The clusters of today are complex, have components with varied computing capabilities, and continue to evolve driven by the applications running on them. The increased complexity has made it difficult to write programs that exploit the computing power of such systems efficiently. As a result, over the last decade, researchers have developed intuitive abstractions and implementations targeting average users, domain experts, and savvy (parallel) programmers.

Among the emerging applications, the most challenging ones to implement efficiently are *irregular applications*. Examples include finding items that are frequently shopped together, simulating the evolution of cosmological bodies, finding nearest-neighbors, quantifying the clustered nature of a set of bodies or points in astrophysics or statistics, and the list goes on. Programs implementing irregular applications operate over data structures such as graphs, trees, and sparse matrices and involve input-dependent computation. In contrast, a regular application such as altering the brightness or contrast of an image operates over a dense matrix and involves input-independent computation, which is often predictable. The predictability of computation makes regular programs amenable to a variety of optimizations applied either automatically by the compiler or manually. Computation in irregular programs, on the other hand, is unpredictable due to the input-dependent nature and hence, optimizations such as parallelizing computation and exploiting locality are either not possible or ineffective when automated techniques through standard compiler transformations are naïvely

applied. Often, irregular programs are used with large data sets that prohibit their execution on a single compute node due to memory limitations on the node. In such scenarios, we need to distribute the computation on multiple nodes and hence, writing distributed-memory programs becomes necessary. This brings with it a host of challenges: firstly, we need to create tasks to generate work for multiple nodes while ensuring that the tasks spend the least amount of time waiting. In other words, task creation should maximize parallelism. Secondly, we need to partition the tasks among different nodes to maximize parallelism and minimize the communication overhead. Thirdly, we should execute the tasks in an order respecting inter-task data dependencies. Finally, we should consider load-balancing among different nodes for improved performance. This explicit management of tasks, communication, and load-balancing is beyond the scope of an average or even a domain expert.

Domain-, algorithm- and architecture-specific insights have led to the development of efficient optimizations and implementations for irregular programs. However, such efforts have been ad-hoc, based on a manual approach, and have largely been the purview of expert programmers. Researchers have developed tools that simplify the creation of distributed-memory parallel programs. Some of these efforts take an automated approach—they *generate* a distributed-memory parallel program starting from some artefact of a shared-memory program—,while others provide efficient abstractions and programming languages.

The key insight in developing intuitive abstractions that simplify the creation of irregular programs is identifying the existence of a latent structure to computation in the underlying algorithms. The abstractions have not only made writing irregular programs accessible to an average programmer but also ensured that the implementations are efficient. The success of graph-processing frameworks such as Galois [1], GraphLab [2,3], Ligra [4], PowerGraph [5], Boost Graph Libraries (BGL) [6,7] etc. is a great example. While graph-based applications have benefited from these frameworks, certain tree-based applications have not. This is counter-intuitive, since trees are kind of graphs. The latent structure of computation in tree-based applications sheds light

into this anomaly: often the trees in are traversed in a depth-first manner covering a large span of the tree and there exist millions or even billions independent traversals. Thus, the absence of a fine-grain parallelism within a single traversal and the inability of existing frameworks to exploit a coarse-grain parallelism effectively makes existing graph processing frameworks unsuitable for tree based irregular programs. When it comes to handling the challenges associated with distributed-memory implementations, in general, there is a trade-off between performance and ease of programming. E.g. BGL is flexible in allowing users to create customized data (graph) partitions. This may result in improved performance due to minimized communication overhead, however, at the cost of users explicitly managing communication. Distributed GraphLab completely hides away the details of data distribution but enforces a certain order of computation which may not deliver the best performance. This trade-off is true in case of full-fledged, general-purpose programming languages such as Legion [8], Parsec [9], Charisma [10], etc. as well. All these approaches still require users to write programs as opposed to automated approaches that focus on tools generating programs. However, automated approaches have had no success w.r.t. creating distributed implementations of irregular programs. This is because of the unpredictability of computation and communication in irregular programs.

Drawing from the success of graph-processing frameworks, can we build abstractions for creating tree-based applications on distributed-memory systems? can we automatically generate distributed-memory irregular programs? can we generalize tree-algorithm specific optimizations to algorithms from other domains? In this thesis, we develop tools to bridge the gap between ease of programming and performance of irregular programs on distributed-memory systems. Specifically, we develop abstractions, optimizations, and systems for creating efficient, distributed-memory implementations of *recursive* irregular programs operating on tree and dense-matrix based data structures.

We look at recursive formulations because they seem natural for a divide-conquer approach to programming. They are also straightforward to reason about and write,

and have properties that make them amenable to optimizations targeting locality and parallelism. In this thesis, we first look at *traversal codes*—a subset of tree-based recursive irregular programs involving repeated tree traversals. We present an abstraction, which simplifies the creation of traversal codes on distributed-memory systems. Next, to better understand tree algorithms and develop optimizations for those algorithms, we consider a broader class of tree programs. We present an ontology and a benchmark suite for tree traversal algorithms. In the final part of the thesis, we continue to look at recursive formulations—now dense-matrix based—and present a tool for automatically generating distributed-memory parallel programs for a subset of divide-conquer algorithms.

Traversal codes appear in an important set of applications based on spatial trees such as kdtrees [11], vptrees [12], octrees [13], and balltrees [14] etc. The trees are traversed depth-first repeatedly, millions or even billions of times corresponding to the input size. As there is an abundant amount of parallelism in these applications due to independence of the tree traversals, a data-parallel approach to parallelizing traversals by executing them simultaneously on multiple tree replicas seems natural. However, the tree sizes grow with data set sizes and for large data sets, there is not enough space to store the tree in memory. So, trees must be distributed across nodes. SPIRIT [15] [1] addresses the insufficiency of traditional data-parallel approaches and existing systems in effectively parallelizing traversals over distributed spatial trees. SPIRIT provides a set of APIs and a runtime system to automate the creation and traversal of distributed spatial trees. SPIRIT partially replicates a tree when space permits. This *space-adaptive* feature allows for a seamless performance transition from a data-parallel approach to purely distributed solution. SPIRIT also employs a host of optimizations targeting locality, parallelism, communication overhead, and load-balance to traverse the distributed trees efficiently.

Whether a new optimization or a tree algorithm, it is challenging to device an efficient implementation strategy considering a rich history of application- and platform-

[1] Scalable Parallel Infrastructure for Recursive Irregular Traversals

specific optimizations and tree algorithms. Treelogy [16] is conceived with the idea of understanding the connection between optimizations and tree algorithms in general. Treelogy provides a categorization scheme (ontology) and a benchmark suite of a broader class of tree algorithms to help answer: (i) is there any existing optimization that is applicable or effective for a new tree algorithm? (ii) can a new optimization developed for a specific tree algorithm be applied to existing tree algorithms from other domains? We show that a categorization based on the structural properties of tree-algorithms is useful for both developers of new optimizations and new tree algorithm creators. We also present a suite of tree algorithms spanning the ontology and implement them on GPU-, shared-, and distributed-memory systems to understand the structural properties of the algorithms.

Given the complexities of distributed-memory programming, is it possible to completely automate the creation of distributed-memory implementations of irregular programs? Here, a broader goal is to let a computer generate a distributed-memory implementation of *any* irregular program starting from its shared-memory artefact (specification, pseudocode, or implementation). D2P is a first step towards this goal. D2P takes as input a specification of a recursive divide-conquer algorithm having certain properties: i) inclusive—a recursive method's parameters summarize the data access done within the method body. ii) Intersection—data-set intersection tests among method invocations can be computed efficiently when a hierarchical decomposition creates disjoint partitions of data, which are computed in a specific order. Recursive formulations of Dynamic Programming (DP) algorithms are well-known examples having these properties. The computation in DP algorithms is not irregular *per se*. However, the communication patterns are irregular. Recursive formulations are also not straightforward for these algorithms. However, they are competitive compared to 'standard', iterative codes and even perform better when implemented carefully. We evaluate D2P on recursive DP algorithms and generate distributed-memory parallel codes capable of exploiting node- and core-level parallelism.

1.1 Contributions

The primary contributions are:

1. SPIRIT, an efficient framework for simplifying the creation of distributed-memory implementations of spatial tree-based applications is introduced. A set of APIs for the creation of and traversal over distributed trees is presented. Experimental evidence is presented showing that SPIRIT implementations i) scale well and can smoothly trade-off space for performance, ii) significantly outperform those done in contemporary distributed graph processing frameworks and are competitive with, and sometimes better than, existing customized, distributed-memory implementations of tree applications.

2. Treelogy, a benchmark suite and an ontology for tree algorithms is introduced. Treelogy consists of a suite of nine tree traversal kernels. The ontology categorizes tree traversal kernels according to their several structural attributes. The kernels span the ontology: for each category, Treelogy has at least two kernels of each type. Experimental evidence is presented showing i) the scalability properties of Treelogy kernels on multiple platforms including GPUs, shared-memory, and distributed-memory systems. ii) traversal algorithms with certain tree types yield better performance compared to the "standard" tree. iii) generalization of an optimization from one application domain to other yields substantial benefits.

3. D2P, an efficient framework for automatically generating MPI+Cilk parallel implementations of recursive divide-conquer algorithms is introduced. Experimental evidence is presented to show that executions of generated implementations scale well, perform significantly faster than those done using similar frameworks, and outperform even *hand-written* distributed implementations in most cases.

This thesis is also the first to categorize and provide a benchmark of tree algorithms. The source code repository of all these works are made freely available to the public.

SPIRIT is published in ICS 2017 [15]. Treelogy is published in ISPASS 2017 [16]. At the time of writing this thesis, D2P is under submission to a conference.

1.2 Outline

The rest of this thesis is organized as follows: Chapter 2 describes novel optimizations and the abstraction that SPIRIT provides. It also describes the evaluation of SPIRIT implementations of five tree applications. Chapter 3 describes Treelogy. It discusses the structural attributes used in categorizing tree algorithms, the mapping of structural attributes to optimizations, and case studies showing the effectiveness of ontology driven optimizations. Chapter 4 describes D2P. It discusses the specification, code generation scheme, and the evaluation of D2P generated implementations of dynamic-programming based algorithms. Chapter 5 concludes our contributions.

2. SPIRIT: A FRAMEWORK FOR CREATING DISTRIBUTED RECURSIVE TREE APPLICATIONS

2.1 Introduction

A common algorithmic pattern that arises in many applications in social networking, data mining, computer graphics, etc., is *tree traversal.* These applications often consist of repeated traversals of a tree—for example, repeated traversals of an object tree during ray tracing to determine the color of each eye ray. These applications tend to possess abundant data parallelism: the tree traversals are independent, so each of the millions or even billions of tree traversals can be performed in parallel. As input sizes increase, the data sets can easily become too large to be processed on a single machine.

A natural approach to continuing to parallelize tree applications is to simply process the data across multiple machines: replicate the tree on multiple nodes, then have each node process a portion of the data set. Sadly, this approach does not really work, because the size of the *tree* being traversed typically increases commensurate with the number of traversals. As a result, it is infeasible to have the tree replicated at each node. Instead, the only possible approach is to distribute the tree across multiple nodes. Consequently, it is necessary to restructure the traversal application to handle traversals that can span multiple nodes.

Tree traversals on a distributed tree presents a host of challenges. First, because the tree is no longer replicated across all of the nodes, we cannot rely on data parallelism arising from processing multiple independent traversals in parallel to provide work for all nodes. We must find parallelism from other sources. One such source of parallelism is *pipeline parallelism*, where each traversal is broken into a pipeline of *sub-traversals*,

and sub-traversals in different portions of the distributed tree can proceed in parallel, allowing all nodes to perform work even if they do not hold the entire tree.

Second, because individual traversals now span multiple nodes, we must schedule tree traversals to maximize locality while minimizing communication. In contrast to the kinds of applications targeted by most distributed and parallel graph frameworks [1, 2, 5, 6, 17, 18], each tree traversal spans large portions of the tree, rather than small regions of a graph. Hence, it is not practical to consider each tree traversal as a single (or even atomic) unit of computation for processing. Instead, traversals must be carefully split up and interleaved to achieve good performance.

Finally, we must ensure that the pipeline does not contain any bottlenecks due to sub-traversals overloading a particular subtree on a node. This requires the ability to identify and replicate bottleneck subtrees on multiple nodes. Because this replication trades off memory usage (more replication consumes more overall memory) for performance (more replication ameliorates more hotspots), an application should be able to tune its replication level to achieve the right balance.

Contributions

In this Chapter, we introduce SPIRIT, a *space-adaptive*, *distributed* tree traversal framework. SPIRIT distributes a tree across multiple nodes, and then carefully schedules multiple tree traversals across these subtrees to achieve parallelism, promote locality, and provide load balance. The chief features of SPIRIT are:

1. SPIRIT provides *pipeline parallelism* for traversals. It splits traversals into sub-traversals that end at subtree boundaries, allowing sub-traversals that are in different subtrees to execute simultaneously on different nodes.

2. SPIRIT carefully schedules the resulting finer-granularity tasks to improve locality: sub-traversals that are accessing the same portion of a subtree will execute in close succession. It also reduces communication: sub-traversals on

a node that want to communicate with the same remote node aggregate their messages.

3. SPIRIT provides load balance by selectively replicating portions of the tree that are heavily loaded, allowing multiple nodes to take on the responsibility of handling those sub-traversals. This feature also allows SPIRIT to be *space-adaptive*: programmers can tune the amount of replication, providing a balance between pure pipeline parallelism, which requires no data replication, but may have limited parallelism in practice, and pure data parallelism, which has abundant parallelism, but requires complete replication of the tree at each node.

4. SPIRIT provides APIs that allows programmers to easily implement their tree traversal algorithms without considering the challenges of distribution or scheduling.

We evaluate SPIRIT on 5 tree traversal benchmarks and show that it achieves both strong and weak scaling across these challenging applications, even when resource limitations prevent fully replicating the tree. We demonstrate the advantages of SPIRIT's space adaptivity—SPIRIT can smoothly trade off space usage for parallelism, and, when sufficient resources are available, a SPIRIT application is able to perform equivalently to a fully-replicated, data-parallel implementation. Finally, we compare SPIRIT implementations to prior work in two ways: we show that SPIRIT implementations far outperform implementations using existing distributed graph frameworks, and we show that SPIRIT is competitive with, and sometimes better than, existing custom implementations of traversal algorithms.

2.2 Background and Motivation

In this section, we begin by discussing terminology and tree traversal characteristics common to the benchmark applications. Scheduling and locality opportunities in a tree traversal algorithm and their applicability in a distributed setting are discussed next.

```
1  void TraverseTree(Vertex v, Point p){
2      status = EvaluateVertex(v, p)
3      if(status is truncate) return
4      if(status is traverse_left){
5          TraverseTree(v.leftChild, p)
6          TraverseTree(v.rightChild, p)
7      }else{
8          TraverseTree(v.rightChild, p)
9          TraverseTree(v.leftChild, p)
10     }}
```

Fig. 2.1.: Tree Traversal in Nearest Neighbor.

Finally, available approaches to distributed graph processing, example frameworks, and the challenges involved in using the frameworks to create distributed tree applications are explained.

2.2.1 Terminology and Traversal Characteristics

This Chapter considers distributing and parallelizing *tree traversal* applications. A tree traversal consists of a set (or multiple sets) of depth-first traversals of a tree (the vertices of the tree can have any number of children, though in practice, most trees are binary trees or octrees). *Points* are the entities that traverse the tree. Points may not perform a complete traversal of the tree: a traversal may be truncated at any vertex, leading the point to skip visiting the subtree rooted at that vertex. While the tree may be altered in the overall application execution scenario, during tree traversal, neither the tree structure nor the vertex properties are altered (in other words, in applications where the tree does change, the changes happen *between* different sets of traversals being performed on the tree). A *node* is a physical machine that houses a partition of the distributed tree and is involved in some computation, which is primarily traversal of points through the partition.

2.2.2 Tree Traversal Algorithms and Locality

Figure 2.1 shows the pseudocode for a nearest-neighbor (NN) algorithm [12] implemented using a kd-tree traversal. The goal is to find closest point(s), based on some distance function, to a query point. First, a set points representing the *feature space* in N dimensions is organized into a kd-tree. The tree is built top-down, recursively partitioning the feature space into two subspaces along one of the N dimensions until a subspace contains a single or some predefined number of points. Each vertex of the tree hence represents a (sub)space of the feature space. The tree is then traversed depth-first to find the closest neighbor(s) of the query point.

The traversal begins by guessing the distance to closest neighbor and sets it to a very large value. At every vertex on the traversal path, the query point determines whether the vertex's subspace *could* contain a point that is closer than the current guess (line 2). If so, the query point updates the current guess and the traversal proceeds to explore the vertex's children. If not, the traversal is truncated and proceeds to unexplored vertices of the tree. When the traversal reaches a leaf, point(s) in the leaf's subspace are inspected and the query point updates its guess for its closest neighbor.

Locality Nearest neighbor applications often have multiple query points and the traversal paths of query points have a set of vertices in common with other query points. This means that traversals of multiple query points can be scheduled to access the common set of vertices in close succession to improve temporal locality. As the queries are independent, the traversals can be interleaved, and multiple vertex computations (by multiple points) can be scheduled simultaneously. Point blocking [19] exploited the locality available at this finer granularity of vertex computations in the context of shared-memory systems.

Distributed traversal A distributed tree traversal must check for the presence of remote vertices—vertices existing on remote nodes. If the vertex is remote, a message must be sent to the remote node for scheduling the *sub-traversal* on the

subtree rooted at the remote vertex and the sub-traversal on the current node must be suspended until the exploration of remote subtree is complete. The current node can then schedule another traversal when available. Traversals of multiple points touching the same remote vertex could be aggregated on the current node before sending a single message to the remote node to schedule their respective sub-traversals. Thus, a *pipeline* of sub-traversals together with aggregation, enable parallel execution of traversals, and reduce communication overhead in addition to improving locality of remote subtree accesses—SPIRIT is designed to leverage this insight.

2.2.3 Graph Processing Frameworks

Many graph processing frameworks allow programmers to write kernels that ignore the distribution of the graph, performing all of the necessary distribution and communication automatically and transparently [1, 2, 5, 6, 17, 18, 20]. However, these frameworks adopt a *vertex-centric* programming model[1]: the kernels that they encourage programmers to write operate on a single vertex and its edges, or, at best, a small region in the graph. Each such kernel is considered a task, and the graph frameworks derive parallelism from processing numerous such tasks in a bulk-synchronous manner. The small size of each task makes it straightforward to schedule for improved locality, reduced communication, task migration for load balancing etc.

For a subset of tree traversal algorithms, the vertex-centric approach is simply not suitable. Traversing an *oct-tree* in Barnes-Hut [21], a *kd-tree* in ray tracing [22] or in computing two-point correlation [23], or a *vp-tree* in finding nearest neighbors [12], is based on depth-first order and hence, there is no parallelism at the vertex level in these traversal algorithms.

Also, the trees are traversed repeatedly without any dependency across traversals. Thus, though there exists plenty of opportunities for parallelism when a sequence of vertex accesses spanning the entire tree is considered, the vertex-centric approach

[1]Parallel BGL [6] and STAPL [20] support coarse-grained computation model in addition to the vertex-centric model.

misses this while focusing on an individual vertex or small region of the tree in identifying independent computations. Moreover, a vertex-centric approach can miss opportunities for better locality or reduced communication by exploiting similarities *across* traversals.

Distributed GraphLab (DGL) [2] DGL adopts a vertex-centric approach to distributed graph processing. Algorithms in DGL are captured in an easy-to-program template of Gather-Apply-Scatter (GAS). A GAS operation on a vertex is essentially a vertex computation that is scheduled for parallel execution by the programmer. In the *gather* phase, results of vertex computations of adjacent vertices are aggregated. The *apply* phase uses the aggregated results to perform some computation at the current vertex. Based on the results of the *apply* phase, the *scatter* phase updates adjacent edges, thereby scheduling new vertex computations on adjacent vertices in the next iterative step. At the end of an iterative step, all the vertices are checked for a change in state to determine convergence. DFS traversals are inherently unsuitable to the GAS model of execution as they have low vertex-level parallelism.

Parallel BGL (PBGL) [6] PBGL provides a more flexible abstraction to graph distribution and traversal: built-in skeletons are provided for creation and traversal of distributed graphs in addition to the data structures necessary for their manual creation. The traversal abstractions (*visitors*) of PBGL are primarily focused on distributing and parallelizing a *single* traversal (e.g., by processing multiple vertices in a breadth-first search simultaneously). Unfortunately, in our tree traversal algorithms, traversals are depth-first, and hence have no opportunity for intra-traversal parallelism. Moreover, PBGL cannot exploit opportunities for *inter*-traversal locality, message aggregation, and load balance that are afforded by the execution of multiple traversals of a single tree.

Thus, to exploit new parallelism opportunities, to schedule traversals for optimizing communication and locality, and to address the challenges in distributed tree creation, we developed SPIRIT, a framework that exploits inter-traversal behaviors to improve

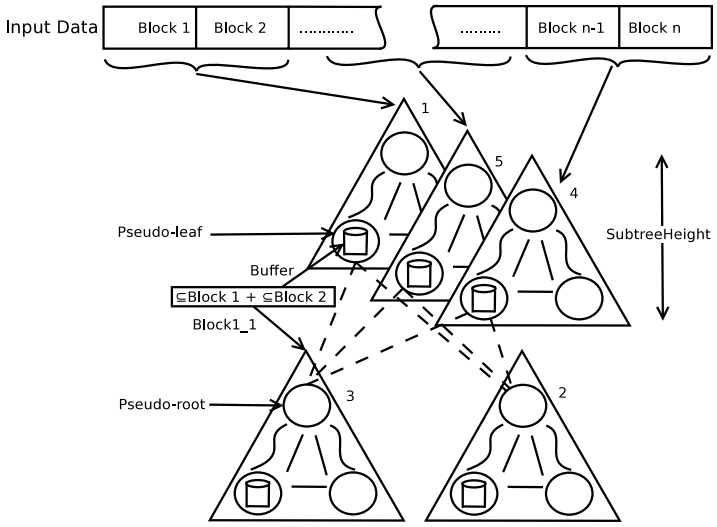


Fig. 2.2.: Tree partitioning and distribution in SPIRIT.

performance and reduce communication. SPIRIT provides a *traversal-centric* set of abstractions for writing distributed tree applications, as described in the following sections.

2.3 Design

This section describes the design of SPIRIT, a framework for writing distributed, space-adaptive tree applications.

2.3.1 Tree Partitioning and Distribution

To distribute computation across multiple nodes, SPIRIT *partitions* the tree into subtrees, with each node assigned one or more subtrees. Figure 2.2 shows a tree partitioned into three subtrees of height, $SubtreeHeight$. The subtrees are labeled 1, 2, and 3, and created on different nodes. (Subtrees 4 and 5 are replicas of 1 and are explained in Section 2.3.4; for now, we ignore them). Each subtree's root, except that of subtree 1, is called a *pseudo-root* (to distinguish them from the true root of

the tree). Similarly, vertices whose children belong to a different subtree are called *pseudo-leaves* (and note that those children are, by definition, pseudo-roots).

SPIRIT includes tree building algorithms for three types of spatial trees: kd-trees, octrees, and vp-trees. Programmers can also implement their own tree building algorithms. Figure 2.3 shows the pseudocode of octree building.

Initially, all the processes collectively determine the bounding box of a 3-dimensional particle space and its center. The root vertex of the tree represents this box. Each particle is then inserted into this box and a vertex may be split into multiple children if the box contains more particles than a user-specified limit. The split results in creation of one or more child vertices corresponding to octant(s) within the enclosing box. If a child is a pseudo-leaf and needs to be split, the tree building procedure halts if the subtree of that pseudo-leaf is remotely owned—$GetSubtreeOwnerID$ returns owner ID of the octants of the enclosing box. Every process now has a top subtree of depth $SubtreeHeight$, connected to a set of local subtrees (in dark) as the tree structure in figure 2.3 shows.

An all-to-all communication of center-of-masses of local pseudo-leaves is performed so that every process has the same vertex data in all the pseudo leaves. The center-of-mass of the vertices in the top subtree is computed bottom up starting from a pseudo-leaf. Kd-tree building follows similar procedure except for the absence of the all-to-all communication. The bounding box and other vertex data needed are updated during particle insertion.

For large data sets resulting in a tree deeper than 2x$SubtreeHeight$ levels, a fixed $SubtreeHeight$ can result in over-fragmentation. SPIRIT avoids this by allowing variable height bottom subtrees. The structure of the octree in Barnes-Hut may change every time step based on the results of traversal from the previous time step. Currently, SPIRIT does not have optimizations in place to re-use parts of the tree that do not require restructuring; entire tree is rebuilt at the beginning of each step.

```
1  BuildOctree(char* input, InputParser* parser){
2     <points, numRead>=DistributeParticles(input, parser)
3     <center, dia>=GetBoundingBox(points, numRead)
4     //Create root vertex and set bounding box
5     <points,numRead> = LoadParticles(input, parser)
6     while(numRead < totalPoints) {
7       for each p in points
8         BuildSubTree(root,p,center,dia,depth)
9     <points,numRead+> = ReadData(input, BATCH)
10    }

12    for each pseudoleaf whose subtree is local
13      computecofm(pseudoleaf)
14    //inform local pseudoleaf info to all processes.
15    for each pseudoleaf
16      all-to-all(pseudoleaf-cofm)
17    //Compute cofm of all vertices in top subtree.
18    ComputeCofm(rootVertex)
19 }
```

```
1  BuildSubtree(Vertex* v,Point* p,float[3] center,float dia,int depth){
2     if(!v){
3       //create a leaf vertex v, insert p into v.
4     } else {
5       //push p to a list cp
6       num=GetOctant(center, p.cord)
7       c=v.child[num]
8       if(c exists and is leaf){
9             if(numPoints contained by cell c <= MAX_POINTS_PER_CELL){
10            //insert p into c's cell
11            return c
12      } else { //split child vertex.
13            //Remove and push all points in c to cp
14            if(c.level == subtreeHeight) {
15            //make c pseudoleaf
16            owner=GetSubtreeOwnerID(c)
17            if(owner != self) return c
18            //otherwise continue recursive insertion of p
19          } } }
20      for each point q in cp {
21        num=GetOctant(center, q.cord)
22        newcen = GetCenter(center,dia,num)
23        v.child[num]=BuildSubtree(c,q,newcen,dia*0.5,depth+1)
24      } }
25    return v
26 }
```

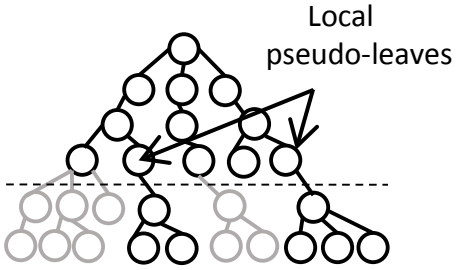


Fig. 2.3.: Distributed octree building.

2.3.2 Tree Traversals and Pipeline Parallelism

For repeated, independent, depth-first traversals of a tree, shifting focus from a point's vertex access to its entire traversal exposes the entire traversal as a target for parallel execution. Because there are numerous traversals in an application, this view of the computation exposes substantial data parallelism: traversals are independent, so the tree can be replicated across nodes, with each node performing a subset of the traversals. However, as the tree size increases with increasing input data size, it may no longer be feasible to replicate the entire tree on each node. As a result, we are forced to distribute the tree across nodes. Unfortunately, this seems to preclude data parallelism! Each traversal needs to visit multiple nodes to complete its work. Interestingly, though, distributing the tree exposes a *new* opportunity for parallelism: *pipeline parallelism*.

Building the pipeline A point's traversal through the partitioned tree can then be viewed as a flow through a pipeline with several, independent stages where each stage corresponds to a sub-traversal visiting one of the subtrees. Because each traversal visits different subtrees (and, indeed, some traversals visit subtrees in different orders), the pipeline stages unfold *dynamically* during traversal. Note that a single subtree can be visited multiple times during a traversal—every stage corresponds to visiting one subtree, but a subtree can be associated with multiple stages. If a point goes from a pseudo-leaf to a pseudo-root or vice-versa, the traversal moves from a pipeline stage associated with the first subtree to a pipeline stage associated with the second subtree. If the subtrees involved are placed on distinct nodes, the traversal then makes a *transition* from one stage to another when the point traverses from the first subtree to the second. For example, in Figure 2.2, a traversal can start at the root of the tree, in subtree 1, then move to subtree 2, return to subtree 1 before proceeding to subtree 3—executing four pipeline stages while touching three subtrees.

Exploiting pipeline parallelism In principle, each (dynamic) pipeline stage is independent of the others, so multiple traversals that are in different pipeline stages can be processed in parallel. In particular, two points that are visiting different subtrees (which may be mapped to different processes) can be processed simultaneously. However, because each process has only one thread (see Section 2.4), in practice, two pipeline stages can only be executed simultaneously if they are in subtrees that are mapped to different processes.

2.3.3 Scheduling for Locality and Aggregation

Stages from different points that visit the same subtree can be co-scheduled to obtain improved temporal locality of vertex accesses. This is achieved through *blocking* of input data: allowing *blocks* of points to traverse the tree simultaneously (in essence, turning the pipeline stages associated with each of the points in the block into a single pipeline stage associated with *all* of the points in the block). Locality is improved by allowing multiple points to interact with each vertex in the subtree consecutively.

Further, when a pipeline stage completes and the block of points is ready to move to another subtree, rather than initiating the next pipeline stage immediately, the block is aggregated in a buffer at the pseudo-leaf. Hence, multiple blocks can traverse a subtree before *all* of the points from those blocks are launched as a single pipeline stage on the next subtree. This *aggregation* further improves locality at the next stage, and also reduces messaging overhead.

Figure 2.2 also shows the blocking of traversal workload, buffers, and the subsets of points from multiple blocks being aggregated in buffers. Note that the buffers contain only subsets of the blocks that traverse the subtree, because some points truncate their traversal and do not arrive at the pseudo-leaf.

Managing block scheduling When some of the points contained within the block truncate their traversals at a vertex, the block continues to traverse with "holes" left at positions of these points. These holes are tracked by keeping *hole maps* or *contexts*,

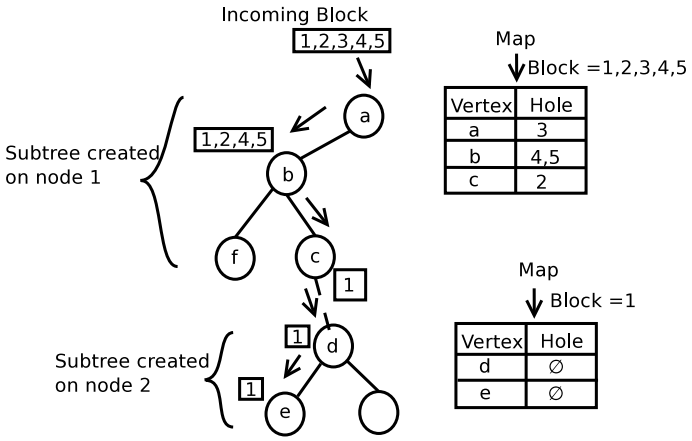


Fig. 2.4.: Block traversal through distributed tree.

vectors at each vertex, marking which holes are created at that vertex (i.e., which points truncate at that vertex). Figure 2.4 shows a block executing a stage associated with subtree 1, followed by a stage at subtree 2, and then executing a stage at subtree 1 again. The hole map on node 1 shows a hole created at position 3 at vertex a, at positions 4 and 5 at vertex b, and so on. No points truncate during the traversal of subtree 2, so the hole maps are empty.

The contexts must be tracked across multiple subtrees: when the block finishes traversing subtree 2 and returns to subtree 1 at vertex c, the hole information for that block must be recalled to ensure that points 1, 2, 4, and 5 are restored to the block so they can all visit vertex f. Tracking contexts efficiently can be challenging; The memory overhead in storing a context is $\mathcal{O}(b+d)$, where b is the size of the block and d is the depth of the subtree.

Because a context can be quite large, SPIRIT does not send it along with the message when the block transitions from one subtree to another. Instead, a block's context is distributed through the tree: when a block traverses a subtree, the context associated with the vertices in that subtree are kept in node-local storage. SPIRIT ensures that when a block returns to this subtree later in its traversal, it returns to the replica on the same node, allowing the context to be retrieved from node-local storage.

2.3.4 Load Balance and Space Adaptivity

Pipelining is critical to achieving good performance. However, the pipelined design as described in previous sections clearly has a bottleneck: all the blocks of input data execute the entry and exit stages associated with a single subtree (top subtree) when they begin and end their traversals at the root vertex. In addition, stages of this subtree are executed whenever a traversal makes a transition from a subtree below. For this reason, the node containing the top subtree remains heavily loaded. To reduce the burden on this node, SPIRIT *replicates* the top subtree across a set of nodes. This *selective replication* ameliorates the pipeline bottleneck without requiring the replication of the entire tree. Figure 2.2 also shows the replicated top subtree on nodes 4 and 5.

Top subtree replication can improve performance greatly as blocks of points can enter the pipeline from any of the nodes where the top subtree exists, allowing multiple nodes to share the burden of processing those blocks.

Space adaptivity In addition to replicating the top subtree, the user can tune the amount of replicated data by also replicating other bottom-level subtrees. Choosing to replicate a subtree effectively eliminates the subtree: the subtree is merged with its (replicated) parent subtree, and the associated pseudo-leaves and pseudo-roots are removed. In this way, the load on that subtree is distributed, and the messaging overhead of communicating across that subtree boundary is eliminated.

Note that if all subtrees are replicated, the entire tree is essentially replicated across all nodes. This immediately yields the data-parallel implementation of a distributed tree-traversal: the nodes are entirely independent of each other, and each node processes a subset of all the traversals on its locally-replicated tree. In this way, SPIRIT is able to smoothly transition between a pipeline parallel implementation (no replication) to a data parallel implementation (full replication) by choosing which subtrees to replicate.

Discussion Given the dynamic nature of traversals, it is difficult to identify any bottleneck subtrees beyond the top subtree. Interestingly, we find that if subtrees beyond the top subtree are replicated, it is not especially important *which* subtrees are replicated: targeting the most heavily loaded subtrees does not perform appreciably better than replicating random subtrees. Section 2.5.3 explores this behavior in more detail.

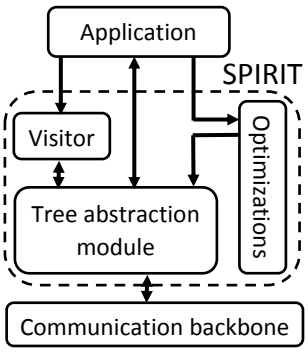
2.4 Implementation

This section describes some of the implementation decisions in SPIRIT and presents an abstraction that can be used to create distributed tree applications.

SPIRIT system Figure 2.5 shows the architecture, interface, and an example of NN implementation in SPIRIT. SPIRIT's implementation uses the wrappers for MPI communication APIs available in BGL. The traversal kernel in SPIRIT is modeled upon BGL's *visitor* pattern. However, a visitor in SPIRIT is designed to take advantage of inter-traversal locality, and message aggregation opportunities, and hence implements a traversal of the tree for a *block* of points, rather than just one. The visitor helps us to separate the details of the distributed traversal of a block from its state; an application programmer overrides a method $EvaluateVertex$, which specifies the vertex computation.

$EvaluateVertex$ returns a status indicating the order in which a traversal wants to visit child vertices in a binary tree. For an octree, we assume that this order is always left-child first and plan to adapt our API to arbitrary orders in future work. Two additional methods need to be overridden: $SetContext$ and $GetContext$. $GetContext$ gets the result of local computation and $SetContext$ updates local state with the result of remote computation.

The *optimizations* module accepts user-configurable values of block size, buffer size, and replication percentage to tune optimizations corresponding to blocking scheduling, aggregation, and replication respectively. Determining an optimal configuration of



SPIRIT* GetInstance(Opts* opt, mpi_process_group& pg)
int BuildKdtree(char* fileName, InputParser* p)
int BuildVptree(char* fileName, InputParser* p)
int BuildOctree(char* fileName, InputParser* p)
int Traverse(Visitor* vis)
Example execution: mpirun -np 4 NN mnist.txt SUBTREE_HEIGHT=8 ...

```
1 enum status{trunc,left,right};
2 int main(int argc, char* argv[]) {
3  //Initialize MPI environment (pg is MPI process group).
4  Optimizations opt(argc, argv);//reads user-specified params
5  InputParser* p= new KdtreeInputParser();
6  SPIRIT* s = SPIRIT::GetInstance(opt, pg);
7  s->BuildKdree(inputFile,p);
8  /*KdtVisitor inherits from Spirit's Visitor class
9  & overrides EvaluateVertex, called from the Traverse API*/
10  KdtVisitor vis();
11  s->Traverse(&vis);
12  s->ResetInstance();
13  return 0;
14 }

16 status KdtVisitor::EvaluateVertex(VertexData* v, Point* p) {
17   if(v.isLeaf) UpdateClosestLabel(p,v);
18   else if(WithinBox(v.boundingBox, p)) {
19   int axis = v->depth % DIMENSION;
20   if(p->coord[axis] < v->splitVal) return left;
21   else return right;
22   }
23  return trunc;
24 }
```

Fig. 2.5.: SPIRIT Architecture, API, and example program

these values is hard. However, a value of block size chosen based on vertex size, subtree height, and available cache size can give good performance [19].

Threads vs. Processes SPIRIT exploits multiple hardware execution contexts (cores or processors) on a single node by mapping multiple independent processes to each node. This design decision allows multiple subtrees to be mapped to a single node, improving load balance through overdecomposition. The processes are distributed

round-robin, first across nodes and then to multiple cores within nodes. For e.g. on 12 processes with a 10-node cluster, we use all 10 nodes for the first 10, map 11^{th} process to 1^{st} node, 12^{th} process to 2^{nd} node and so on. Processes can either be chosen in round-robin (default) or random order to map them to subtrees. and adjacent subtrees whose pseudo-roots have the same parent are mapped to the same process. Blocked (default) or block-cyclic distribution of workloads to top-level subtrees is supported.

In order to alleviate the memory pressure arising from the need for each process to maintain its own storage for input data, SPIRIT uses shared memory segments: all the processes mapped to a node share the memory segment. Due to OS limitations, on some systems we cannot allocate all the points in a single shared memory segment to share between processes. In such situations, we process the input points in batches. While this introduces overhead that limits scaling (as we must impose a barrier between the batches), this is purely an artifact of OS limitations and not a limitation of SPIRIT's design.

An alternative to exploiting multiple cores is to parallelize the processing of a single subtree within a process, handing out multiple blocks (or sub-blocks) to multiple threads. This multi-threading approach would have worse load balance compared to the multiple-process approach, and implementation-wise, would require another level of parallelism to be uncovered. SPIRIT uses the multi-process approach both because load balance is a substantial problem in pipeline parallelism and because it is easier to scale-up with existing implementation infrastructure to utilize more cores within a node.

2.5 Evaluation

This section evaluates SPIRIT on five tree applications from the domains of scientific computing, data mining, and computer graphics. We begin by presenting strong-scaling and weak-scaling numbers for each of these benchmarks (Section 2.5.2), demonstrating the effectiveness of SPIRIT's combination of pipelining and selective

replication. We then evaluate the space-adaptive capabilities of SPIRIT, showing how SPIRIT can smoothly trade off space usage (i.e., replication amount) for performance. Next, we present a performance breakdown consisting of the impact of subtree height, aggregation, and process utilization. Finally, we compare SPIRIT implementations of our benchmarks to implementations in two general graph frameworks—distributed Graph Lab and PBGL—as well as against existing, state-of-the-art distributed implementations of specific applications.

2.5.1 Methodology

We evaluate four versions of SPIRIT:

i **Spirit**$_{DP}$: The data-parallel version of SPIRIT.

ii **Spirit**: The space-adaptive version of SPIRIT replicating only the top subtree.

iii **Spirit**$_{PO}$: **Spirit** with pipeline-parallelism only (no subtree replication).

iv **Spirit**$_{RO}$: Same as **Spirit**, but no pipeline parallelism (i.e., a subtree traversed by a block is "locked out" until that block completes its traversal of the entire tree; some parallelism is still available because there are multiple replicas of the top subtree).

Benchmarks The benchmarks are taken from the Treelogy suite [16]. Spirit is publicly available under the distributed-memory implementations of Treelogy benchmarks at: `https://bitbucket.org/plcl/treelogy`.

i textbfBarnes-Hut (BH) [21]: is an efficient algorithm to simulate interaction of gravitational bodies. In our tests, we present the performance of an octree implementation run for 10 simulation steps.

ii **Nearest Neighbor** (NN): is described in section 2.2.

Table 2.1.: Data sets; |V|= Vertex set size, |P|=Number of points.

Input	Description	Benchmark		
		Name	$\|V\|$	$\|P\|$
Mnist	Handwritten digits data with reduced dimension to $\mathbb{R}^7$	PC, NN, VP	2x|P|-1	10^6
Plummer	Data in plummer model, $\mathbb{R}^3$	BH	≈1.5x|P|	10^6
Christmas	Wavefront .obj file	PM	462,818	10^6
Synthetic	uniformly distributed data in $\mathbb{R}^3$(BH), $\mathbb{R}^2$(PC,NN,VP)	PC,NN, BH,VP	=2x|P|-1 =|P|(VP)	{16,32,64, 128}x10^6

iii **2-Point Correlation** (PC) [23]: is used to determine how clustered a data set is. The goal is to find the number of pairs of points that are within a specified distance, R, from each other. SPIRIT includes kd-tree implementation of PC.

iv **Vantage Point** (VP) [12]: is a nearest neighbor algorithm using vp-trees, where the partitioning plane is a hyper-sphere instead of a hyper-plane as in a kd-tree.

v **Photon Mapping** (PM) [24]: is an algorithm to realistically simulate interaction of light with objects in a scene. The objects in a scene are represented as triangle mesh, whose coordinates are then organized into a kd-tree. Ray-object intersection tests are accelerated through the depth-first traversal of the kd-tree.

Workloads and distribution Table 2.1 describes the data sets (real [25] and synthetic) used in our experiments. PC and BH workloads are inherent to the data set sizes. For other benchmarks, workloads are due to batching of multiple user requests: NN and VP workload sizes are half the data set sizes and PM's workload is due to tracing 1 million rays through a 480x620 pixel scene. The workloads are not pre-processed with techniques such as sorting. For the largest inputs used, the virtual memory requirement for the trees ranged from 10GB(VP)-20GB(PC), and for the total workload (including traversals) ranged from 32GB-48GB. In all the evaluated versions, message aggregation is turned on since the performance is better than that without aggregation. Block size is fixed at 4096, and the buffer size is set to the block size. Unless otherwise noted, all performance measurements are taken on tree

traversal times (exclusive of tree building time, as the traversal time is the dominant cost), and the runtimes are measured using wall-clock time. Every configuration of a test is run until a steady state is achieved, which yields errors of ±1% of the mean with 95% confidence. The locality optimized baselines represent single node runtimes and employ block scheduling [19]. The block size that gave the best performance (not necessarily the same block size in each configuration) is chosen; these implementations hence represent the best-available single-process implementations of each benchmark.

Platform and program development environment All experiments are run on a 10-node cluster with gigabit ethernet interconnect. Each node contains two 10-core, Intel Xeon-E5-2660-v3 processors and runs RHE Linux release 6. A core has 32KB L1 data and instruction cache, and 256KB of L2 cache. The cores share 25MB L3 cache and 64GB main memory. All benchmarks are implemented using C++ and Boost Graph Libraries (BGL 1.55.0) [2]. The programs are compiled using mpic++, a wrapper compiler for gcc 4.4.7, and linked with mpich2-1.4.1p1 [3] and the corresponding BGL wrapper library functions.

2.5.2 Scalability

When SPIRIT is given enough space to replicate the tree, it essentially runs in data-parallel mode, and achieves excellent scaling ($Spirit_{DP}$ plots). The purpose of our scalability experiments is to explore the behavior when SPIRIT *cannot* replicate the tree across all inputs. In such a setting, SPIRIT's pipelining allows it to achieve speedup in both strong-scaling and weak-scaling settings. While this speedup is far less than when SPIRIT is able to fully replicate the trees, SPIRIT's performance is on par with existing hand-written, optimized, application-specific implementations (Section 2.5.5), despite its generality.

[2]http://www.boost.doc/org/libs; PBGL version is same as BGL as PBGL libraries are an extension of BGL.
[3]http://www.mcs.anl.gov/mpi/mpich2

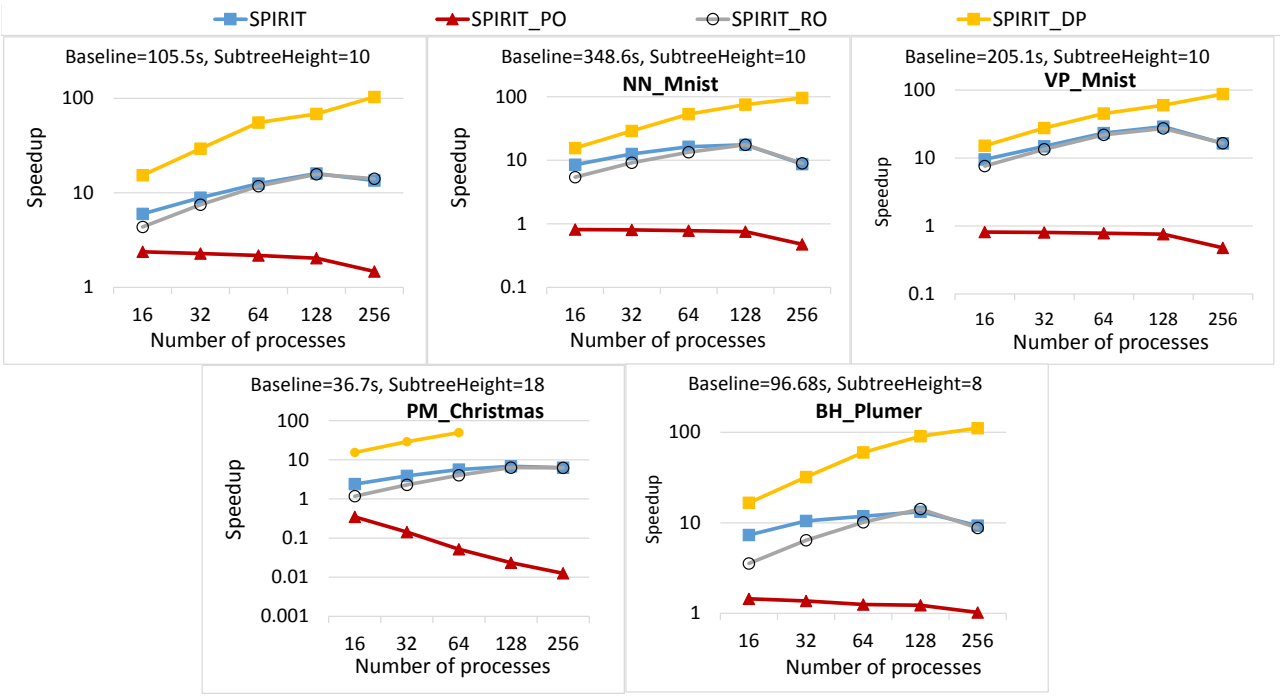


Fig. 2.6.: Strong scaling in SPIRIT.

Strong-scaling We evaluate strong scaling in SPIRIT with both synthetic and real inputs but present here only representative results with real inputs due to lack of space. The input sizes used here are constrained by inputs that allow single-process runs to complete in a reasonable amount of time.

Figures 2.6 shows the results. Overall, *Spirit* scales well and outperforms $Spirit_{PO}$, and $Spirit_{RO}$. $Spirit_{DP}$ evaluations of PM at 128 and 256 did not run because of memory constraints. Across all the benchmarks, the geomean speedup in *Spirit* is 16.3 and that in $Spirit_{RO}$ is 7.7. Recall again that while this speedup seems small, it is in a setting—distributed trees—where, absent SPIRIT's pipelining and load balancing, we would not expect speedup at all.

The $Spirit_{PO}$ plots confirm the proposition of section 2.3.4 that pipeline-parallelism alone is not sufficient to obtain good performance and that selective data replication is necessary. It may seem that the top subtree replication enabling data-parallel execution of sub-traversals is the only reason for *Spirit*'s superior performance. However, comparison between $Spirit_{RO}$ and *Spirit* plots sheds further light: pipeline-parallelism

yields, on an average, 1.54x increase in the speedup performance of $Spirit_{RO}$: the combination of pipelining and replication provide the best performance.

As the input size is fixed, the per-process number of blocks of input decreases (for a fixed block size) with increasing number of processes. As a result, available pipeline-parallelism decreases. Hence, the performance difference between $Spirit_{RO}$ and *Spirit* reduces as the number of processes increase. Note that in $Spirit_{RO}$, sub-traversals execute in parallel only on the replicated top subtrees and there can only be one block in flight on bottom level subtrees. However, for an application like NN or VP, there still exists some pipeline-parallelism in $Spirit_{RO}$ because of the existing intra-block parallelism: a block may split into two sub-blocks at every vertex, each of which can be independently scheduled. This makes a single block traversing through the top subtree spread into multiple bottom subtrees, each of which execute the resulting sub-blocks in parallel. In other words, even with pipelining ostensibly turned off in $Spirit_{RO}$, pipeline parallelism is still exploited.

There is more intra-block parallelism, hence pipeline-parallelism, in NN than VP (traversals in VP are much shorter). As a result, $Spirit_{RO}$ does not yield as much pipeline parallelism in VP, and turning on both replication and pipelining produces a big difference in performance. Because of the lower communication overhead in VP, the absolute speedup numbers are greater than those in NN while producing the same result. BH offers the maximum benefits due to pipeline-parallelism because of the presence of more pseudo-leaves for a given top subtree height. We find that the difference between $SPIRIT_{RO}$ and *Spirit* is even more with larger data sets used in the weak scaling experiments.

Weak-scaling Because our target applications are irregular, it is challenging to ensure that the amount of work done per node is fixed as the input and system size scale up. Instead, we scale the input size with the number of nodes—this results in more traversals and larger trees. Note that these benchmarks are $O(n \log n)$ applications,

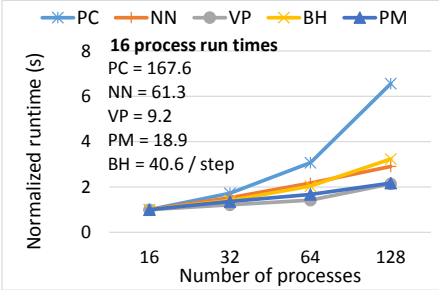


Fig. 2.7.: Weak scaling in SPIRIT.

so a linear increase in the input size results in a larger increase in the amount of work; in this experiment, we expect the overall execution time to increase.

As figure 2.7 shows, we essentially see that behavior. Here, the input sizes are too big to fully replicate the tree, so only a distributed approach such as in SPIRIT suffices to execute these inputs. Every process performs 0.5 million traversals. The tree size varies approximately from 16 to 128 million vertices. Tree traversal times normalized against 16-process runtimes are measured.

VP shows the best weak scaling. This is expected because of the shortest average traversal length causing minimal communication overhead. PC shows the worst weak scaling. This is because of a fixed radius value: the need for finding all pairs of points within a fixed radius causes disproportionate increase in traversal length. PM also shows flat scaling. This is due to the tree size in PM experiments being constant. Due to the unavailability of inputs (scenes) with large number of object triangles and the difficulty in generating synthetic input specifying geometry and other properties of objects in scenes, we scale-up the rays being traced from 0.5 million to 4 million while keeping the tree size constant.

2.5.3 Space-adaptive Evaluation

Figure 2.8 shows the space-adaptive capability of SPIRIT. Given a simple strategy of replicating *random* subtrees (in addition to the top subtree), SPIRIT supports space-adaptivity by varying the *number* of such random subtrees. Selecting more subtrees

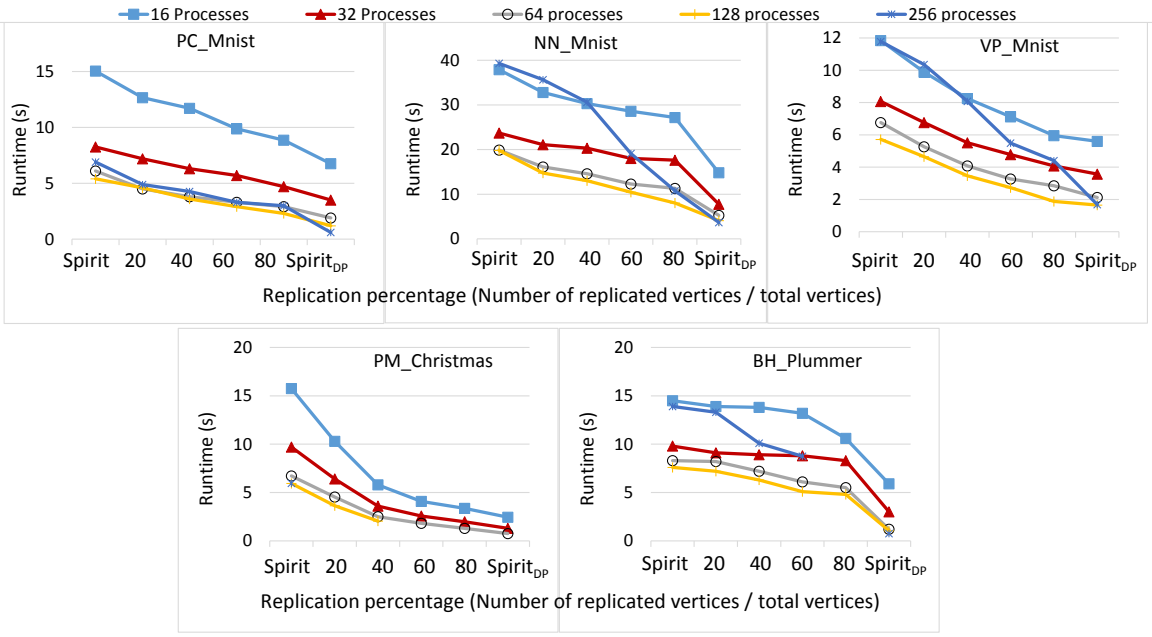


Fig. 2.8.: Space adaptivity in SPIRIT with varying amount of replication

to replicate gives more replication, and hence better performance, at the cost of more space usage. The x-axis represents the user-specified percentage of total vertices of the tree that are replicated. The left-most data point corresponds *Spirit* configuration and the right-most is $Spirit_{DP}$, which involves no communication and hence, represents the best performance. Partial replication beyond the *Spirit* configuration in PM was not possible in case of 256 processes and hence, the corresponding plot is not shown. As we see, in all cases, increasing the amount of replication improved performance.

Replication serves two purposes: mitigating load-imbalance in the pipeline and reducing messaging overhead. The top subtree is by far the biggest bottleneck in the pipeline, as all traversals must visit the top subtree but, due to truncation, lower-level subtrees may not be visited by all traversals. Hence, the load balancing benefits of replication come primarily from replicating the top subtree. Further replication primarily serves to eliminate pseudo-leaves in the top subtree and hence messaging: if a lower-level subtree is replicated, then there is no need to perform communication

Table 2.2.: Process utilization: S=Dataset Size, P=Number of processes, TT=Total traversal time (seconds), VT=per-process computation time as a percentage of total traversal time, D=aggregate volume of data exchanged in MBytes.

S	P	PC			NN			VP			BH			PM		
		TT	VT	D	TT	VT	D	TT	VT	D	TT	VT	D	TT	VT	D
16M	16	167.6	74.6	291.1	61.3	9.6	693.6	9.2	34.6	122.8	40.6	31.6	617.6	18.3	8.4	307.5
	32	88.6	61.4	302.8	42.9	7.2	720.6	5.8	30.3	128.1	25.5	25.8	640.3	11.6	6.7	316.2
	64	56.3	41.8	307.9	39.1	4.1	733.8	4.3	22.3	130.5	17.5	18.6	650.7	8.4	4.9	320.6
	128	37.9	31.1	310.5	34.3	2.6	740.3	3.9	12.4	131.6	15.5	10.9	655.9	7.2	3.3	323.1
128M	16	3671.5	79.3	2467.6	317.2	25.3	2923.5	98.3	42.8	942.9	458.7	43.8	4940.7	87.1	18.5	1218.7
	32	2277.5	76.1	2565.8	196.1	21.5	3046.4	58.2	40.5	984.2	260.8	36.3	5122.4	47.5	16.5	1289.2
	64	1291.5	70.1	2609.4	148.1	15.4	3102.8	38.9	32.6	1002.2	179.1	26.2	5206.1	29.5	13.5	1420.5
	128	1099.5	41.8	2630.8	97.3	12.3	3130.3	32.2	20.5	1010.9	145.2	16.3	5247.6	42.6	11.1	1749.6

when transitioning to its (former) pseudo-root, as both the subtree and its parent are on the same node.

An interesting consequence of this dichotomy (replicating the top subtree eliminates the bottleneck, while replicating other trees primarily targets reduced communication) is that it does not particularly matter *which* subtrees (beyond the top one) are replicated: any replicated subtree eliminates a pseudo-leaf, and while some subtrees are visited more than others, this is a second-order effect.

We confirm this behavior by evaluating a version of SPIRIT that replicates *bottleneck* subtrees, rather than random subtrees: we perform a profiling run to determine which subtrees are the most heavily loaded, and replicate those. We find that the performance difference between this profile-driven strategy and the random strategy is negligible. We also find that dynamic subtree replication strategies perform worse compared to replication during tree construction. It turns out that SPIRIT's simple strategy is sufficient.

Overall, we note that we get the best performance with fully data-parallel implementations. It is important to note, though, that this requires replicating the entire tree across every node, which may consume too much memory. Overall, these experiments demonstrate that SPIRIT is able to adapt its space usage to the resources at hand, and use pipeline parallelism to mitigate the loss of data parallelism when full replication is not possible.

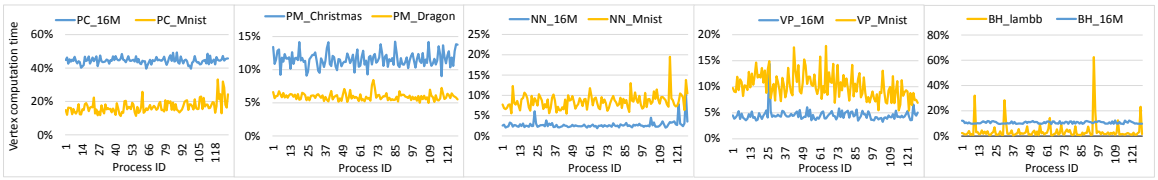
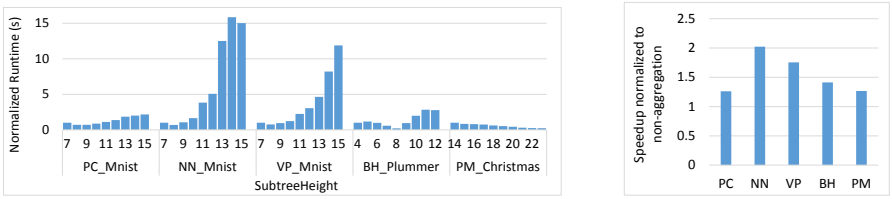


Fig. 2.9.: Load distribution among 128 processes.



(a) (b)

Fig. 2.10.: Impact of a) Subtree height. b) Aggregation.

2.5.4 Performance Breakdown

Process utilization Table 2.2 shows the process utilization measured as the ratio of computation time over total traversal time. We show the results for the smallest and largest of the synthetic inputs. Computation time (VT) is obtained from the average of computation times of all processes. The processes are either performing local computation or incurring overhead: either from actively communicating with other processes to handle remote traversals or polling for messages that indicate the completion of remote traversals. In other words, VT represents the "real" computation performed by SPIRIT, as opposed to overheads incurred due to distribution. We expect to see two trends: first, as more processes are added for a fixed input size, we expect VT to decrease as a percentage of traversal time, as more time must be spent in handling distribution. Second, if the input is increased for the same number of processes, we would expect relatively more time to be spent in VT, as each process holds a larger portion of the tree, and hence performs more computation. Table 2.2 confirms both of these trends.

Load imbalance Figure 2.9 shows the load distribution with respect to both synthetic and real inputs. The figure shows, for each process id in the 128-process run, the relative amount of time spent in computation vs. messaging overhead (in other words, a higher number means more time spent performing real computation). If there were load imbalance, we would expect to see that some processes spend significantly more time in computation (vs. messaging overhead) than others.

As can be seen from the figure, SPIRIT's subtree based partitioning and distribution scheme results in uniform load among all processes mostly. However, BH_lambb is an exception (presence of peaks) as lambb is a clustered data set. Hence, some subtrees of the octree are visited more frequently during the force calculation stage. The current distribution scheme of allocating all the sibling subtrees to a single process results in overloading this process when the sibling subtrees are frequently visited. For this particular input, we found that distributing the tree differently resulted in better load balance (nevertheless, all of the results shown in this section use SPIRIT's default distribution).

Subtree size and message aggregation Figure 2.10a shows the impact of subtree size and aggregation on the performance of 128-process runs. As $SubtreeHeight$ is a rough indicator of subtree size, we measure the performance with varying subtree heights. As $SubtreeHeight$ is increased, the number of pseudo-leaves increases, which increases pipeline-parallelism and also the messaging overhead. So, we expect the performance to improve initially and then degrade as the overhead outweighs parallelism benefits. The figure shows this trend for all the benchmarks except in case of PM. The kd-tree distribution in PM results in decreasing number of pseudo-leaves with increasing height due to the long and narrow tree structure. This improves performance due to more replication and less overhead. Figure 2.10b shows the speedup improvement due to message aggregation, over non-aggregated runs with all inputs used. Overall, aggregation yields a 1.6x speedup improvement. NN and

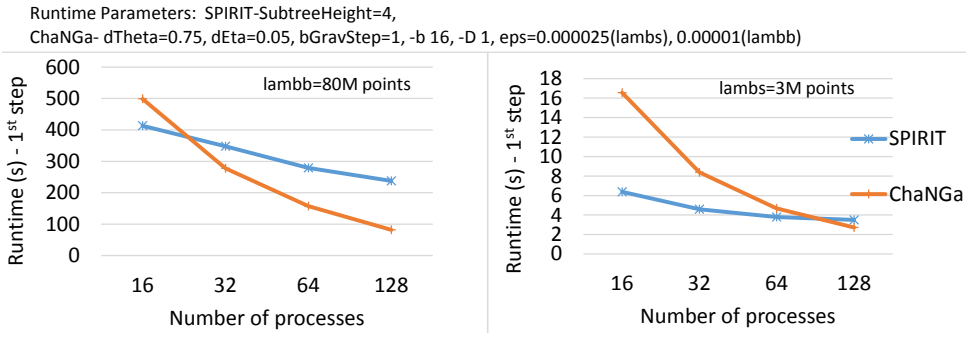


Fig. 2.11.: SPIRIT vs. ChaNGa.

VP show higher benefits compared to other benchmarks because of the improved pipeline-parallelism resulting from increased intra-block parallelism.

2.5.5 SPIRIT, DGL, PBGL, and Reference Software

In this section, we compare *Spirit* with implementations using one traversal-centric and one vertex-centric framework:PBGL and DGL. We also compare *Spirit* with ChaNGa [26], which is the state-of-the-art distributed implementation of Barnes-Hut (reference software for distributed implementations of PM and NN were not available). Since *Spirit* is specialized for distributed tree processing, we expect *Spirit* to perform better in comparison with generic distributed graph processing systems like PBGL and DGL. However, since *Spirit* lacks the application-specific customizations that reference software systems are able to exploit, we might expect *Spirit*'s performance to lag behind that of ChaNGa.

The DGL model is not the right programming fit for any tree benchmark involving depth-first traversals: the executions continue to run beyond 4 hours—due to excessive synchronization (once after every vertex computation)—and hence time out. However, the traversals in DGL are implemented easily as state transitions encapsulated in GAS template.

PBGL traversals employ asynchronous message-passing and avoid excessive synchronization. However, the traversals visit more vertices than necessary, and do not

optimize communication. Implementing PBGL traversals requires a little more effort than DGL: defining a $DFSVisitor$ class and overriding its methods $discover\_vertex$ and $finish\_vertex$, and passing an instance of the class and other bookkeeping arguments (for tracking vertex visits) to the $tsin\_depth\_first\_visit$ API. The API allows a programmer to traverse a distributed tree without worrying about the underlying tree distribution. However, the API does not handle multiple traversals efficiently, and has no support for enforcing the order of vertex visits—which ruled out PM implementation. These factors cause more vertex visits and hence, more work to be done. Also, the API does not take advantage of increased data replication when available and this creates a pipeline bottleneck as in $Spirit_{PO}$.

Spirit, due to efficient truncation handling, optimizing communication, and avoiding the pipeline bottleneck, outperforms PBGL significantly (>150x geomean speedup). Also, *Spirit* implementations have the least average lines of code (90) compared to DGL (200) and PBGL (192) implementations as they encapsulate coarse-grained computations (entire traversal) similar to PBGL and fine-grained computation (vertex computation) similar to DGL.

SPIRIT vs. ChaNGa Figure 2.11 compares the strong scaling performance of SPIRIT and ChaNGa. These tests are conducted on real inputs [27]. Overall these experiments show that SPIRIT is competitive with application-specific implementations.

SPIRIT's treecode computes only Newtonian gravity forces while ChaNGa is more general. In order to ensure that both ChaNGa and SPIRIT compute the same result, the performance impact due to softening force calculations in ChaNGa is mitigated by setting very low eps values. ChaNGa contains many application-specific optimizations for incremental computation that allow it to run faster over multi-step simulations by avoiding re-computing forces that have not changed since the previous time step. However, the first computation step represents an apples-to-apples comparison between ChaNGa and SPIRIT: both fully compute the forces for each body. In this first step,

SPIRIT is actually 1.2 to 2.6 times faster than ChaNGa for all inputs at smaller number of processes.

We attribute the faster performance of SPIRIT to its model of moving computation to data, rather than ChaNGa's model of moving data to computation—fetching and caching remote subtrees on node-local storage when a traversal visits them. At larger number of processes and larger scales, the overhead due to batch processing of input traversals as mentioned in section 2.4 brings down SPIRIT's performance. In addition, SPIRIT's default subtree distribution policy incurs some load imbalance issues that could be addressed with a different distribution (see Section 2.5.4) We plan to address these issues in the future.

2.6 Related Work

Data partitioning, optimizing communication, overlapping computation with communication, and exploiting locality is critical to achieving high-performance in distributed irregular applications. While the performance of such applications has been analyzed in detail [28–30], creating high-performance distributed-memory implementations remains a challenge. Distributed-memory graph programming frameworks partially automate and hence, simplify the creation of such applications. Frameworks based on the vertex-centric model [1, 2, 5, 17, 18] are widely used in implementing such applications. However, they are not suitable for certain tree applications due to the 'bulk-synchronous' model of computation causing excessive communication and the absence of vertex-level parallelism in the tree applications. Frameworks with a flexible programming model [6, 20, 31] allow a coarser-granularity of computation to be captured and employ asynchronous message-passing to minimize the communication overhead. In fact, Nguyen et. al. [29] showed that bulk-synchronous model applied to vertex-centric formulation of some algorithms does not always give high performance and asynchronous scheduling along with algorithm-specific optimizations are necessary. While some of these frameworks do not aggregate messages [2, 6] some do [17, 20].

However, existing locality opportunities due to coarse-level computation granularity are missed. SPIRIT exploits locality, optimizes communication, and automates traversals in a subset of tree applications. In doing so, SPIRIT adopts a traversal-centric approach, aggregates messages, and employs asynchronous scheduling in creating high performance implementations of distributed tree applications.

High performance, distributed implementations of irregular applications [26, 32–34] exploit application-specific knowledge such as how the tree changes from iteration-to-iteration in Barnes-Hut [26], and the range of bin size in two-point correlation [32]. SPIRIT being general, currently does not have these optimizations incorporated. However, these application-specific optimizations could be incorporated into existing implementations.

Distributed kd-trees are constructed bottom-up [33, 35], as against top-down, to avoid touching all points at every level. The domain-decomposition is customized for distributed octree construction in BarnesHut [26]. SPIRIT adopts a top-down, *iterative* approach for kd- and octree construction to avoid touching all the points in input data set at once. In the Hadoop implementation of distributed kd-tree by Aly et. al. [35], the top subtree height is limited by the number of nodes. SPIRIT exploits pipeline paralllelism by making the height user-configurable. In replicating the top subtree, SPIRIT generalizes a Barnes-Hut specific optimization of locally essential trees [36] (replicating the top subtree captures only a subset of the interaction lists), and that of subtree caching of ChaNGa.

There has also been much work on application-specific and independent [19, 34, 37] scheduling approaches to exploit locality in tree traversals. While these techniques are proposed in the context of shared-memory systems and rely on data-parallel formulations, SPIRIT adopts an application-independent technique [19] in a distributed-memory setting.

2.7 Conclusions

We presented SPIRIT, a framework for creating distributed tree traversal applications. SPIRIT provides algorithms for tree distribution, supports distributed execution of applications when the input cannot be replicated on all nodes. It provides pipeline parallelism to expose parallelism in the distributed application. SPIRIT uses a block-scheduling scheme and message aggregation to maximize locality and reduce messaging overheads. Finally, it uses selective replication to avoid bottlenecks in the pipeline, allowing programmers to tune the amount of replication in an application to trade off performance and space usage. Our evaluation showed that SPIRIT can (strong- and weak-)scale across five benchmarks, and that its space adaptivity allows for increasing performance when more space is available for replication. Finally, we showed that SPIRIT implementations far outperform implementations in generic graph frameworks, and that SPIRIT provides competitive performance compared to reference software.

3. TREELOGY: A BENCHMARK SUITE FOR TREE TRAVERSALS

3.1 Introduction

Applications in a number of computational domains including scientific computing [38, 39], computer graphics [24], data mining [12, 40–42], and computational biology [43], are built around *tree traversal* kernels, which perform various computations by traversing trees that capture structural properties on input data. Because these tree traversals are time consuming, memory intensive, and complicated, there has been substantial interest in developing optimizations and implementation strategies that target improving the performance of tree traversal [26, 36, 40, 44–48].

While tree traversal kernels are widespread, they are also highly varied in terms of the types of trees that are used (e.g., octrees, kd-trees, ball trees, etc.), the traversal patterns of those trees, the number and variety of separate traversals that are performed, etc. Each new tree algorithm requires careful thought to determine which implementation strategies and optimizations are likely to be effective, and hence developers of new tree-based algorithms may struggle to devise efficient implementations.

Conversely, developers of optimizations targeting tree traversals also face challenges in determining how effective and applicable their optimizations are. Unfortunately, existing graph benchmark suites feature only a handful of tree traversal kernels [49–54], not providing enough variety to understand the generality and behavior of new optimizations. Tree traversals are a distinct subclass of graph algorithms, with their own unique challenges—traversals touch large portions of highly-structured tree data, unlike most graph kernels which operate on small, localized *neighborhoods* of the graph [55]—so evaluating implementation strategies and optimizations for tree kernels requires benchmarks that cover the breadth of tree traversal behaviors.

Interestingly, many optimizations depend on particular *structural* characteristics of tree traversal kernels to be effective: some optimizations target only top-down, pre-order traversal kernels [48], while others work for top-down traversals but not for bottom-up traversals [44]. Others rely on a fixed traversal order of the tree [46], while still others allow traversals to traverse the tree in any order [47]. However, there has not been any attempt to identify this set of characteristics in a manner that allows multiple optimizations to be targeted to particular tree traversal kernels. Much past work on tree traversals was aimed at efficient expression and optimization of specific tree traversal kernels [24, 26, 40, 45, 56]. Understanding these characteristics, and how they affect optimization opportunities, is critical to optimizing tree traversal kernels.

3.1.1 Contributions

To better understand traversal algorithms, and develop and understand optimizations for those algorithms, it is helpful to have a set of benchmarks that span a wide range of characteristics. To that end, this paper presents *Treelogy*, a benchmark suite and an ontology for tree traversal algorithms.

1. We present a suite of nine algorithms spanning several application domains: (1) Nearest neighbor [12]; (2) K-nearest neighbor [12]; (3) Two-point correlation [40]; (4) Barnes-Hut [38]; (5) Photon mapping for ray tracing [24]; (6) Frequent item-set mining [41]; (7) Fast multipole method [39]; (8) K-means clustering [42]; (9) Longest common prefix [43].

2. We develop an *ontology* for tree traversal kernels, categorizing them according to several *structural* attributes. Treelogy kernels span the ontology: for each category, Treelogy has at least two kernels of each type.

3. We present a mapping of existing tree traversal optimizations to the types of traversals described by our ontology, and show how the ontology can guide which optimizations can be applied to which kernels and vice-versa.

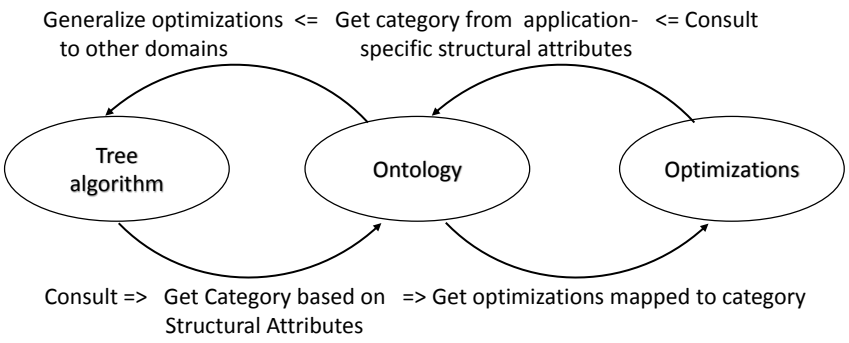


Fig. 3.1.: Treelogy use case.

4. We evaluate the benchmarks in Treelogy with multiple types of trees, on real and synthetic inputs, and across multiple hardware platforms: GPUs, shared memory, and distributed memory. Using our evaluation framework, we present results showing the scalability of reference implementations of the benchmarks, and demonstrate that traversal algorithms with certain tree types yield better performance compared to the "standard" tree.

Treelogy thus benefits both the designers of new tree algorithms as well as the developers of new tree traversal optimizations, as shown in Figure 3.1. If a designer devises a new tree algorithm, he or she can use Treelogy's ontology to quickly categorize the algorithm and determine which existing optimizations and implementation strategies are likely to yield an efficient implementation. If a developer produces a new optimization for tree traversals, he or she can use the ontology to help determine what structural properties are necessary for the optimization to apply, and use the benchmarks of Treelogy to evaluate his or her optimization against reference implementations. Treelogy is publicly available at: `https://bitbucket.org/plcl/treelogy`.

3.1.2 Outline

The remainder of this paper is organized as follows. Section 2.2 presents background on tree traversal algorithms, including a discussion of spatial trees and a general

skeleton for tree traversal algorithms. Section 3.3 describes the kernels of Treelogy. Section 3.4 presents our ontology for tree traversal algorithms, maps the kernels of Treelogy to the ontology, and discusses how this ontology can be used to determine the suitability of different optimizations. Section 3.5 evaluates the kernels of Treelogy on multiple inputs and on multiple platforms. Finally, Section 3.6 summarizes related work and Section 3.7 concludes.

3.2 Background

This section describes spatial indexing structures that organize the input data in the form of the trees that underpin the tree traversal algorithms. Next it discusses the structure of an example tree traversal kernel and optimizations that are common to most of Treelogy's benchmarks (i.e., those that do not depend on deeper structural characteristics).

3.2.1 Trees for Accelerating Computations

The use of trees to optimize different types of computations is common across many algorithms. These trees often take one of two forms: *spatial acceleration structures*, that organize data in an n-dimensional metric space, and *n-fix trees* (our term that covers both prefix and suffix trees) that organize sets of sequences according to similarity.[1]

Spatial Acceleration Structures

As Gray and Moore argue, an effective, general way to speed up the computation of n-body algorithms is through the use of spatial trees [40]. An n-body computation, in its naïve implementation, requires comparing each of a set of items with every other item in a data set. Rather than performing this $O(n^2)$ process, spatial trees organize

[1]We do not mean to imply that these are the only types of trees; merely that they are among the most common.

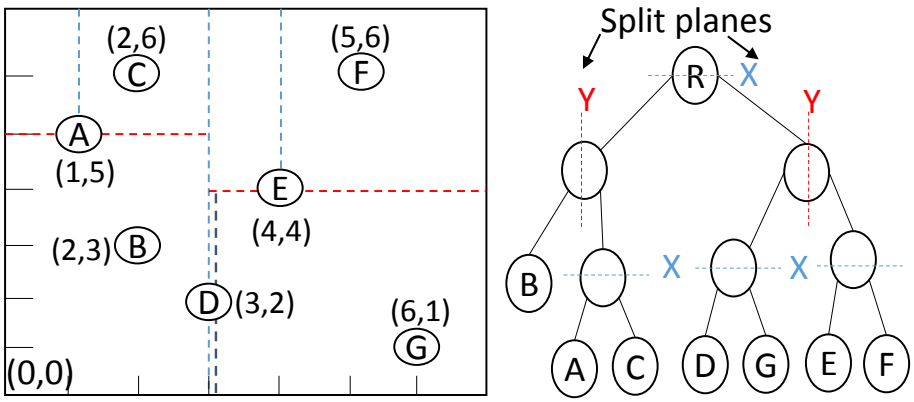
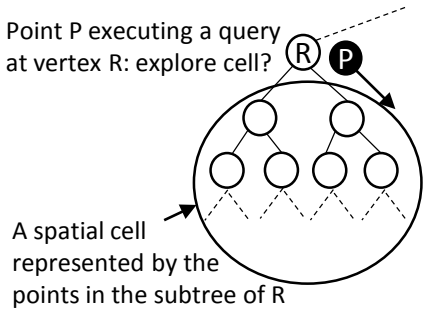


Fig. 3.2.: Sample space (2D) and corresponding kd-tree.

the items of the set into a tree structure to speed up the comparison process. These trees work for data that is embedded in a metric space (or a pseudo-metric space, where a distance can be determined between any pair of points).

For example, a kd-tree [11] organizes k-dimensional spatial data (hence the name kd-tree) by recursively splitting the set of points into subspaces by cutting the space using a split plane along one of the dimensions to divide the points in the current space in half. Each split subdivides the space into two, and the two subspaces are made children of the parent space in the kd-tree. In this way, the tree is built top down, with the root of the tree representing all of the points, and the leaves of the tree representing a single point (or a small set of points). Figure 3.2 shows an example of a kd-tree built in two dimensions over 7 points. This structure allows for a very fast proximity check between any point p and the entire subspace represented by a node in the kd-tree: if any part of the subspace is "close" to p, that means that the subspace may contain a point within that distance of p. Because most n-body codes are fundamentally concerned with the question of which points are close to each other (whether to compute a 2-point correlation, or estimate forces, etc.), this query allows the tree to be traversed, and in doing so, quickly eliminate entire subspaces without individually visiting points (see Figure 3.3a).



(a) Query - is any point in the cell close enough to P? if no, don't explore cell

```
1  void Traverse(Vertex v, Point p, float radius)
2  {
3      if(distance(v, p) > radius)
4          return;
5      if (v.isLeaf())
6        foreach (Point p1 : v)
7          if (distance(p1, p) < radius)
8            p.corr++;
9      else
10       Traverse(v.leftChild, p, radius);
11       Traverse(v.rightChild, p, radius);
12 }
```

(b) Two-Point Correlation

```
1  void Traverse(Vertex v, Point p){
2      status = ComputeVertex(v, p)
3      if(status is truncate)
4          return
5      if(status is traverse_left){
6          Traverse(leftChild, p)
7          Traverse(rightChild, p)
8      }
9      else{
10         Traverse(rightChild, p)
11         Traverse(leftChild, p)
12     }
13 }
```

(c) Nearest Neighbor

Fig. 3.3.: Purpose of spatial acceleration structures and example traversal codes

So, for example, to use a kd-tree to compute the two-point correlation of a point p, where the purpose is to determine how many points are within a radius r of p, the point p starts at the root of the kd-tree. As long as the current cell of the kd-tree overlaps with any portion of the (k-dimensional) sphere around p, the point traverses down the tree. Otherwise, that entire subtree (and hence subspace) can be truncated from the traversal. If p reaches a leaf of the tree, it compares with the point(s) in that leaf. In this way, p's 2-point correlation can be computed without comparing to every other point in the space. Figure 3.3b shows pseudocode for this traversal. Other n-body computations can be performed in similar ways [40].

Kd-trees are one of many types of spatial-acceleration trees. Others include vp-trees (vantage point trees) [12] where subspaces are determined not by being on one side or another of an orthogonal split plane, but instead by being inside or outside hyperspheres. BSP trees (binary space partitioning trees) [57] use split planes to divide subspaces just like kd-trees, but those planes may not be orthogonal. Octrees [13] for 3-dimensional spaces, as used in Barnes-Hut, do not evenly divide subspaces by the distribution of points but instead by distance: when a cell is divided into subspaces, the subspaces are eight equally-sized subspaces. Quad-trees are similar to octrees and are for 2-dimensional spaces. Unlike kd-tree or octree, sibling subspaces in Balltree [14] are allowed to intersect and need not partition the whole space. As we see in Section 3.5, the type of spatial tree used affects the behavior of each traversal, and hence performance.

N-fix Trees

Another common type of tree is used to organize data that is not embedded in a metric space, but instead shares another form of similarity: common sequences. For example, prefix trees (tries) are a space-efficient structure for storing a set of strings, with nodes in the tree representing letters, and paths in the tree representing words. Two words that share a common prefix (e.g., “cat” and “car”) share the same path

in the tree for their common prefix ("ca-"), and then split into two children for their suffixes. Suffix trees are similar, but organized in the opposite direction. While these trees are often used for efficient representations of strings (suffix trees are especially common for representing sets of genomic sequences [58]), they also are used for other sets, such as in frequent item-set mining [41].

3.2.2 Traversal Structure and Optimizations

Efficiently implementing point correlation (Figure 3.3b) requires considering many issues. Note that a point's traversal touches the vertices of the tree in depth-first order resulting in no scope for intra-traversal parallelism if the traversal order is to be preserved. However, more often, multiple points traverse the tree, these points are independent of each other, and the tree is not modified during traversal. Hence, there exists ample coarse-grained parallelism. In such a scenario, when multiple traversals touch a set of vertices and perform computation at vertices in the set (referred to as vertex computation from now on), more than one traversal may touch the same vertex. Therefore, traversals can be reordered in such a way that those touching similar vertices can be scheduled in close succession so as to enhance temporal locality. Block scheduling [59] is a generic locality enhancement technique where a *block* of points traverses the tree rather than a single point, allowing multiple points to interact with a single vertex from the tree.

While the previous paragraph described the properties of the traversal kernel and optimizations possible for point correlation only, we find that many tree-based traversal kernels share these properties. In fact, we can systematically categorize tree traversal kernels so that kernels in the same category share similar properties and hence can be reasoned about collectively when they are the focus of optimizations. Section 3.4 describes this ontology.

3.3 Treelogy Traversal Kernels

This section presents the 9 traversal kernels of Treelogy: *Nearest Neighbor* (NN), *k-Nearest Neighbor* (KNN), *Two-Point Correlation* (PC), *Barnes-Hut (BH)*, *Photon Mapping* (PM), *Frequent Itemset Mining* (FIM), *Fast Multipole Method* (FMM), *k-Means Clustering* (KC), and *Longest Common Prefix* (LCP).

1) ***Nearest Neighbor*** [12] is an optimization problem in data mining, image processing, and statistics, where the goal is to find closest point(s), based on some distance function, to a query point. Figure 3.3c shows pseudocode for an NN implementation. First, a set of points representing the *feature space* are organized into a metric tree. A query point then traverses the tree depth-first to find its closest neighbor(s). The traversal begins by guessing the distance to closest neighbor and sets it to a very large value. At every vertex on the traversal path, the query point determines whether the vertex's subspace *could* contain a point that is closer than the current guess. If so, the query point updates the current guess and the traversal proceeds to explore the vertex's children. If not, the traversal is truncated and proceeds to unexplored vertices of the tree. When the traversal reaches a leaf, point(s) in the leaf's subspace are inspected and the query point updates its guess for its closest neighbor. Typically, there exist several query points and hence offer scope for parallel execution. Treelogy includes NN implementations using kd- and Vantage Point (vp) trees.

2) ***K-Nearest Neighbor*** finds the k nearest neighbors to each query point [60]. Instead of chasing the only nearest neighbor, KNN maintains a distance buffer to record the K closest neighbors. KNN is more robust to noisy training data than NN, and the k-neighbor buffer gives it a significantly different traversal pattern than NN. Treelogy includes KNN implementations using ball-trees and kd-trees.

3) ***Two-Point Correlation*** [40] The traversal algorithm in two-point correlation was introduced in section 3.2.1. This is an important algorithm in statistics, and data mining used to determine how clustered a data set is. As in the case with NN,

different types of trees are possible for input representation. Treeogy includes PC implementations using kd- and vp-trees.

4) ***Barnes-Hut*** [38] is an efficient algorithm to predict the movement of a system of bodies that interact with each other. BH accelerates the computation of forces acting on a body due to its interaction with other bodies. It does this by building an octree (3-dimension) based on the spatial coordinates of those bodies. Every body then traverses the tree top-down to compute the force acting upon it. The force due to far away bodies can be approximated based on the center of mass of those bodies, allowing direct body-to-body interactions to be skipped. The algorithm runs for multiple time steps. At the end of each time step, the bodies' positions are updated based on the computed forces, and the tree is rebuilt using the updated positions. Treelogy includes BH implementations using octree and kd-trees. Our kd-tree implementation is based on the recursive orthogonal bisection method [28].

5) ***Photon Mapping*** [24] is an algorithm to realistically simulate interaction of light with objects in a scene. A kd-tree is used to accelerate the ray-object intersection tests. The objects in a scene are represented by triangle meshes. The triangle coordinates are then organized into a kd-tree structure. Each of a set of input rays traverses the kd-tree depth-first to determine the triangles (objects) that it intersects with. The algorithm proceeds in multiple phases, with each phase generating a set of reflected and refracted rays, which in turn traverse the tree in subsequent phases. The algorithm terminates when there are no reflected or refracted rays. Our implementation is adapted from HeatRay[2].

6) ***Frequent Item set Mining*** [41] is a data mining kernel typically employed in mining associations and correlations (e.g. finding correlation in the shopping behavior of customers in a supermarket setting). The input is a set of transactions, T, each containing a subset of items from the set B, which contains all the items that are available for purchase. Another input, $supp_{min}$, quantifies the term "frequent". The

[2]`https://github.com/galdar496/heatray/`

goal is to return all sets of items, I, that are "frequent" i.e. $\forall b_i \in I, |C(b_i)| \geq supp_{min}$. Where, the cover of b_i, $C(b_i) = T_i | (T_i \in T, b_i \in T_i)$

A naïve algorithm of generating all possible item sets (*candidates*) and then scanning the transaction set, T, is infeasible since it requires generation of $2^{|B|}$ item sets, which is impractical for large $|B|$ values, and T is often too large to fit in memory. Hence, modified prefix trees [41] are used to compactly represent T in memory. The tree is then systematically traversed in a bottom-up manner to generate the sets of all frequent items occurring in some combination in any transaction. This process involves generating additional *conditional* prefix-trees that are iteratively traversed. The FIM implementation in Treelogy is adapted from FPGrowth[3].

7) ***Fast Multipole Method*** [39] is an efficient algorithm to speed up the computation of particle interaction forces in an *n-body problem* e.g. computing potential of every particle in a system of charged particles. The spatial coordinates of particles are organized into an octree or quad-tree. The tree is then traversed in three steps (top-down, breadth-first in first step, bottom-up, depth-first in second and third steps) to compute the potential of all particles. We characterize the performance of top-down (TD) and bottom-up (BU) traversals separately in section 3.5. Our quad-tree FMM implementation is based on "low-rank approximation of well-separated regions" [61].

8) ***K-means Clustering*** [42] is a popular cluster analysis method in data mining. It partitions a large number of data points into K different clusters. The algorithm works in an iterative way. First, every point computes the distance to these K clusters, and is assigned to the closest one. Then every cluster is updated to be the average position of the points that belong to it. This process repeats until all the points belong to the same clusters in two successive iterations.

While most implementations of K-Means are non-tree based, Treelogy implements a kd-tree based version of the algorithm [42]. For each iteration, a spatial tree is build to organize the set of clusters. Then the point traverses the tree to look for its nearest cluster. Because of the tree structure, a point can quickly filter out far away clusters

[3] `https://github.com/integeruser/FP-growth/`

to avoid unnecessary distance computations. The tree-based K-Means usually presents a better performance [42]. Our implementation is adapted from KdKmeans[4].

9) ***Longest Common Prefix*** [43] or longest common substring (LCS) problem is common kernel in bioinformatics and document retrieval. LCP finds the the longest string that is a common substring of two (or more) strings. Given two strings with length N and M, while the dynamic programming method typically takes O(N*M) time, a suffix tree-based traversal can solve the LCP problem in O(N+M) time. The longest common prefix of both strings can be found by building a generalized suffix-tree, and finding the deepest internal nodes that contain substrings from both input strings. Our implementation is adapted from Longest Common Substring[5].

3.4 An Ontology for Tree Traversals

This section introduces an *ontology* for kernels in tree applications, identifying five key features that help categorize tree traversal kernels. We then explain how the categorization of a kernel according to these features can help direct which optimizations apply to a given kernel.

3.4.1 Ontology

1. **Top-down and Bottom-up:** Top-down traversals perform a traversal of the tree beginning from the root. Bottom-up traversals traverse the tree from leaves to the root. Top-down traversals can be preorder, inorder or postorder. There is a correspondence between postorder and bottom-up traversals in an algorithmic implementation: bottom-up traversals can be implemented as postorder. However, some optimizations only apply to top-down traversals [47, 62], while bottom-up traversals can avoid some of the recursion overhead of postorder

[4] https://github.com/vaivaswatha/kdkmeans-cuda
[5] http://www.geeksforgeeks.org/generalized-suffix-tree-1/

traversals, making the choice of one or the other significant depending on the situation. In our experience, most tree traversal kernels are top-down.

2. **Unguided and Guided:** A top-down recursive traversal of a tree, whether preorder or postorder, effectively linearizes the tree. In many applications, every traversal of the tree creates "compatible" linearizations: while a single traversal may not visit the whole tree (due to truncation), there exists a single linearization of the tree where every traversal's linearization is a subset of that canonical linearization. For example, in a top-down traversal where every traversal visits a node's left child before visiting its right, all traversals have compatible linearizations. In other applications, though, one traversal might visit the left child before the right while another might visit the right child before the left. In this case, the two linearizations don't match: they are not subsequences of the same canonical linearization. Borrowing terminology from Goldfarb et al. [48], we call the former type of traversal *unguided* and the latter *guided*.

   Crucially, in an unguided traversal, the traversal order of the tree is not traversal dependent. In a guided traversal, however, not only is the traversal order of the tree dependent on properties of a given traversal, that order *could change based on computations performed during the traversal.* Figure 3.3b shows an unguided traversal: any point in 2-point correlation will traverse the tree in the same order. Figure 3.3c shows a guided traversal: which order a point traverses the tree depends on the results of `ComputeVertex`.

3. **Type of tree:** While the traversal kernels of Treelogy can be implemented with a variety of spatial and n-fix trees, this paper presents efficient implementations using oct-, k-dimensional (kd), quad-, ball-(bt), vantage-point(vp), suffix-, and prefix-trees. Various considerations, including memory usage, structural balance and input characteristics can influence the selection of tree types for an algorithm, and can lead to differing performance. Sometimes the tree type is constrained

by the dimensionality of input space: 2-dimensional input in FMM requires a quad-tree while 3-dimensional input is represented with an octree.

4. **Iterative with tree mutation:** In iterative applications, the tree is traversed repeatedly until a terminating condition is satisfied. Traversal kernels in such applications modify the tree structure between successive iterations. However, within an iteration, the tree structure is not modified, thus allowing multiple traversals to execute simultaneously. Tree structure modification affects distributed-memory implementations, where tree building and distribution can consume a significant amount of time.

5. **Iterative with working set mutation:** In traversal kernels of iterative applications, the number of independent traversals executing simultaneously may vary across iterations. This set of independent traversals is referred to as the working set. This varying size of the working set can translate to opportunities in load-balancing, and often in minimizing parallel overheads due to synchronization.

Ontology Applied to Treelogy

The benchmark programs in Treelogy span the ontology: for each attribute type, Treelogy includes at least two benchmarks covering each possible value for that attribute (with the exception of tree type, as tree type is often independent of algorithm). Table 3.1 shows the benchmarks classified according to our ontology. Note that most of Treelogy's algorithms are top-down traversals of various kinds. This is consistent with our experience that most tree-traversal kernels are top-down.

3.4.2 Optimizations

Over the years, researchers have proposed numerous optimizations for tree traversal kernels. This section presents specific optimizations, and elucidates how the ontological

Table 3.1.: Classification of benchmarks

Benchmark	Traversal order	Traversal guidance	Tree type	Tree mutation	Work set mutation
NN_kd	preorder	guided	kd	×	×
NN_vp	preorder	guided	vp	×	×
KNN_kd	preorder	guided	kd	×	×
KNN_bl	preorder	guided	bl	×	×
PC_kd	preorder	unguided	kd	×	×
PC_vp	preorder	guided	vp	×	×
BH_oct	preorder	unguided	oct	✓	×
BH_kd	preorder	unguided	kd	✓	×
PM_kd	preorder	unguided	kd	×	✓
FIM_pre	bottom-up	unguided	prefix	✓	✓
KC_kd	inorder	guided	kd	✓	×
LCP_suf	postorder	unguided	suffix	✓	×
FMM_qd	preorder and bottom-up	unguided	quad	✓	×

characterization of a tree traversal can be used to determine whether an optimization is generalizable.

Locality Traversals through the top part of the tree represent a negligible amount of work compared to traversals through the bottom part. In many top-down traversal kernels, the behavior of traversals in the top part of the tree can provide insight into behavior in the bottom part. Hence, by profiling traversals in the top part, better temporal locality can be achieved through optimized scheduling of traversals through the bottom part [47, 62]. Such profiling is ineffective in case of post-order or bottom-up traversals due to large volume of profiling data. For bottom-up traversals, a tiling optimization such as that in cache-conscious prefix trees [56] can be effective in enhancing spatial locality.

Vectorization An optimization for Barnes-Hut [63] that is used in vectorized implementations involves a pre-processing step which linearizes traversals before doing any vertex computation. Similarly, many GPU implementations of tree traversals rely on pre-computing *interaction lists*: the tree is traversed to determine which nodes of the tree are touched by each traversal, and only afterwards are the actual computations performed [46]. These types of optimizations can only be applied for unguided traversals: they require either a single linearization of the tree, or that the linearization can

be computed without performing the full traversal. These optimizations cannot be applied for guided traversals such as NN, since the next vertex to be visited depends on the result of current vertex computation. In such a scenario, in order to achieve vectorization, the tree structure needs to be modified with other techniques such as autoroping and lockstepping [48].

Input representation Bottom-up traversal of suffix trees was shown to be efficient in performing multiple genome alignment [64]. Depth-first traversals through prefix trees in FIM beat breadth-first traversals through subset trees [65]. In NN, vp-trees were shown to be the best option compared to alternate input representation options in case of handling queries for finding similar patches in images [45].

In case of spatial trees, traversal properties through one type of tree can be very different when compared to traversals through other types: while vp-trees can facilitate faster truncation of a traversal—hence shorter traversals and faster performance—insertion and deletion is difficult, and hence may not be a good choice for a kernel involving frequent updates to the tree structure. A kd-tree typically offers a height-balanced structure, but at the cost of more vertices, while an octree represents the data more compactly but is not usually balanced. As we see in section 3.5, NN, PC, and BH implementations with certain types of trees yield faster traversal performance for certain inputs.

Other optimizations Locally essential trees (LET) [36] is a BH-specific optimization. When the input domain of cosmological bodies is decomposed and represented as a tree structure, nearby bodies in a spatial subdomain see (and traverse) a similar subtree structure because the fine- and coarse-grained interactions of these bodies with bodies in other subdomains are similar. In order to compute the total force acting on each body in that subdomain, the union of all such subtrees is needed. This union is referred to as an LET. Essentially, by replicating portions of the tree, the tree structure necessary to compute forces is made available locally to a processor's domain (in a distributed-computing scenario).

Distributed-memory implementations of kernels involving repeated, top-down traversals benefit from the generalization of this optimization: the top subtree (necessary for a subset of point-cell interactions) can be replicated on node-local memory to achieve scalability and minimize inter-node communication.

Estimating returns of optimizations We provide a simple reuse distance analyzer tool [66] to estimate the efficacy of locality optimizations in tree traversals. Reuse distance is known to be a good predictor of cache performance, despite its many simplifying assumptions. Our tool analyzes locality at the granularity of vertices of a tree, so does not provide complete reuse distance analysis but an analysis tailored to our suite. Reuse distance analysis can help provide architecture-independent measures of locality: small reuse distances mean good locality, large reuse distances mean poor locality but, importantly, an opportunity to potentially improve locality. Hence, the analyzer helps us understand the potential for optimization in a benchmark and/or input. Section 3.5 provides a case study of using the tool to verify the effectiveness of locality optimization.

An ontological characterization of tree traversals provides better guidance to design code transformations and optimizations focusing on performance. We have presented a rigorous classification, well known classes of optimizations and a methodology to estimate the returns of a particular class of optimization for tree traversals. Next to support our design, we provide an evaluation of our benchmark suite in different aspects.

3.5 Evaluation

We now present the evaluation of our benchmark kernels.

3.5.1 Methodology

We evaluate the following variants of tree traversal kernel implementations:

1. **SHM**: shared-memory implementation that is run on a single compute node.

2. **DM**: distributed-memory implementation that is run on a cluster of nodes, using the approach of Hegde et al. [67].

3. **GPU**: GPU implementation, using the approach of Liu et al. [62].

Since the goal of this study is to characterize traversal kernels, we do not profile the entire application. Hence, the performance measurements show the traversal times only. Especially, for GPU implementations, we only measure the computation kernel runtime spent on GPU. All our baselines are single-threaded *SHM* versions that optimize scheduling of multiple independent traversals by executing *blocks* of traversals simultaneously [59]. All *DM* runs execute a data-parallel configuration of the kernel, where the tree is replicated on each node and the set of independent traversals are partitioned among the nodes of the cluster. Every configuration of a test is run until a steady state is achieved, which yields errors of 1.2% of the mean with 95% confidence.

In addition to the optimized variants we evaluate here, the Treelogy distribution includes unoptimized, single-threaded implementations for use as optimization targets.

Data sets We evaluate each benchmark on both real-world[6] and synthetic inputs. We use a publicly available tool for generating FIM data[7], and also provide synthetic data generators for each benchmark that allow users to create inputs of the desired size and dimensionality, allowing them to evaluate the behavior of implementations at different input sizes. Table 3.2 shows the details of the inputs used.

Platform and Program Development Environment

A single compute node consists of 20 Xeon-E5-2660 cores with hyperthreading and runs RHE Linux release 6. Each core has 32KB of L1 data and instruction

[6]Source: https://www.ncbi.nlm.nih.gov/genome/viruses/, UCI Machine Learning Repository
[7]FIM: https://sourceforge.net/projects/ibmquestdatagen/

Table 3.2.: Data sets and attributes; |V|= Number of vertices, |P|=Number of traversals.

Input	Description	Benchmark		
		Name	$\|V\|$	$\|P\|$
Synthetic1	Uniformly distributed data in 7-dimensional space ($\mathbb{R}^7$)	PC_kd, NN_kd	2x10^7-1	10^7
		PC_vp, NN_vp	10^7	10^7
Mnist	Handwritten digits data with reduced dimension to $\mathbb{R}^7$	PC_kd, NN_kd, KNN_bt	2x10^6-1	10^6
		PC_vp, NN_vp	10^6	10^6
Plummer	Plummer model with initial position, velocity, and mass	BH_kd BH_oct	2x10^6-1 ≈1.5x10^6	10^6
Synthetic2	Uniformly distributed data, $\mathbb{R}^3$	BH_kd BH_oct	2x10^7-1 ≈15x10^6	10^7
Christmas	Wavefront .obj file	PM_kd	462,818	2.3*10^6
Dragon	Wavefront .obj file	PM_kd	22,532	1.4*10^6
Gazelle2	KDD Cup-2000 data set	FIM_pre	202,234	202,234
Synthetic3	IBM Quest Data Generator	FIM_Pre	23,301	23,301
Synthetic4	Uniformly distributed data, $\mathbb{R}^2$	KC_kd	128	204,800
Wiki	Word vectors from first 10^8 words of wikipedia, $\mathbb{R}^{100}$	KC_kd	128	71,291
Synthetic5	Uniformly distributed data with constant mass and potential, $\mathbb{R}^2$	FMM_qd	≈10^6	≈10^6
Synthetic6	Strings each of size 10^5 construted randomly using an alphabet of 7 characters	LCP_suf	296,007	NA
Genome	Origin of two viral genome sequences from NCBI	LCP_suf	10,132	NA

Table 3.3.: Runtime characteristics of benchmarks.

Benchmark	Input	Baseline time (s)			Average traversal length	C_v	L3 miss rate (%)	CPI
		SHM	DM	GPU				
PC_kd	Syn1	926.4	1066.1	29.5	13790	0.13	65.15	1.82
	Mni	79.5	80.2	2.6	2606	0.38	14.74	0.75
PC_vp	Syn1	553.8	422.6	199.0	296	0.14	70.68	3.84
	Mni	2.7	3.2	0.8	40	0.001	40.21	1.67
NN_kd	Syn1	1250.4	1580.9	60.2	3143	0.14	48.2	1.39
	Mni	124.3	180.4	19.4	6310	0.31	16.8	0.63
NN_vp	Syn1	2815.5	4512.8	280.0	3234	0.21	58.9	2.66
	Mni	87.1	107.5	8.6	2657	0.26	13.6	1.01
KNN_bl	Syn1	2424.1	3284.3	106.8	3808	0.74	39.81	1.3
	Mni	145.6	237.2	4.3	3955	1.74	15.3	0.82
BH_oct	Syn2	841.08	2200.7	1.4	1403/step	0.11	63.6	4.32
	Plu	57.86	149.9	2.9	2709/step	5.86	35.55	1.59
BH_kd	Syn2	278.4	953.7	0.3	818/step	0.1	47.52	1.88
	Plu	62.1	169.68	1.3	7704/step	1.92	21.1	0.64
PM_kd	Chr	30.4	49.3	NA	93/step	0.78	42.92	2.07
	Dra	5.1	9.54	NA	86/step	0.68	4.3	1.19
FIM_pre	Gaz	3.9	4.0	NA	2	7.3	2.37	1.17
	Syn3	23.9	24.1	NA	2	1.5	19.4	1.48
KC_kd	Syn4	25.3	16.9	3.7	11/step	0.09	6.42	0.67
	Wiki	98.03	91.9	41.5	128/step	0.02	21.6	0.58
LCP_suf	Gen	0.05	0.02	NA	10,132	NA	17.17	0.84
	Syn6	1.27	0.63	NA	296,007	NA	58.24	1.01
FMM_qd	Syn5							
	TD	5.7	5.3	0.03	64/step	0.19	0.7	0.46
	BU	0.6	0.2	NA				

cache, and 256KB of L2 cache. The cores share 25MB of L3 cache, and 64GB of RAM. The *DM* experiments are run on a cluster of 10 such compute nodes. *DM* implementations are compiled using mpic++—a wrapper compiler over gcc 4.4.7—and linked with Boost Graph Library (BGL 1.55.0) and MPI libraries (MPICH2 1.4.1p1). *SHM* implementations are compiled using gcc 4.4.7, and linked with *pthread* libraries. GPU implementations are compiled with NVCC 7.0.27, and evaluated on a server with 32 GB physical memory, two AMD 6164 HE processors and a nVidia Tesla K20C GPU card (5GB memory on board). The K20C deploys 13 Streaming Multiprocessor (SMX) and each SMX contains 192 single-precision CUDA cores.

Metrics

We first characterize the Treelogy benchmarks according to various architecture dependent and independent metrics. Table 3.3 shows the results.

Average traversal length For each benchmark/input pair, we compute the *average traversal length* of the traversals in the kernel. The length of a traversal is measured as the number of vertices that the traversal touches in the tree (note that we count each vertex just once, even if a traversal performs work at a vertex during both pre-order and post-order portion of the algorithm). Longer traversal lengths means that more of the tree is being touched, and that traversals have larger working sets. This has implications for locality (larger working sets means more likelihood of cache misses) and for scheduling (longer traversals, especially in relation to the size of the tree, means that there is more likelihood that traversals overlap and hence can be scheduled together for better locality or smaller divergence).

The L3-cache miss rate results measured on the baseline SHM performance in Table 3.3 reflect this: we see higher miss rates with longer traversal lengths and larger inputs. However, PC_vp in comparison with PC_kd is a contradiction; despite the traversals through vptree being much shorter (for the same input), we see an increase in miss rate. This is because of the guided traversal kernel of PC_vp offering more intra-block parallelism compared to the unguided, PC_kd kernel, which means that for the same block of input points, a larger portion of the tree is touched in case of PC_vp. All other cases of lower miss rates are due to very shorter traversal lengths (PM, FIM, FMM, KC) and smaller tree sizes (KC, PM_Dra).

The following observations from Table 3.3 emphasize how input-dependent the kernels are: a) while kd-tree offers the best traversal performance for Syn1, vptree offers the best performance for Mni for NN benchmark. From another perspective, traversals through the kd-tree are the shortest for Syn1 but longest in case of Mni when compared to vptree and ball-tree traversals. b) we also see that while every traversal in KC_kd touched the entire tree for Wiki, only 11 vertices, on average, were touched in Syn4. In summary, the behavior of these benchmarks is highly dependent on the tree type and even input distribution.

Load-distribution To determine how the work of a traversal algorithm is distributed across the tree (useful for distributed memory load balancing), we measure *load-distribution*. We partition the tree into subtrees such that the resultant tree is two subtrees deep (i.e. the tree is logically sliced at half the maximum depth). The amount of work done by a subtree is the load, which is roughly equivalent to the sum of the size of all blocks executed at all vertices of the subtree. Since the lone top subtree has the maximum load because of all traversals beginning and ending at the root vertex, we skip this subtree from our analysis to get a clearer picture of load on other subtrees. For measuring load-imbalance, we calculate $C_v = \sigma/\mu$, where μ is the average load on a subtree and σ is the standard deviation. C_v, the coefficient of variation, indicates the presence of subtrees that are heavily loaded. C_v value for LCP is not measured as this is a single postorder traversal kernel. While a very small value of C_v indicates uniform load, a large value indicates the presence of bottleneck subtrees. Load distribution matters for implementations of traversal benchmarks that distribute the trees [26, 67], since different sub-trees may have different computational loads, necessitating load balance strategies such as replication of bottleneck subtrees.

We expect that the overall tree structure and the input distribution influence the load on a subtree. The C_v values in Table 3.3 reflect this. The higher values of C_v for BH with plu input is because of the clustered nature of the data set. In case of KNN, PM and FIM, the higher value is mainly because of the resulting fragmented tree struture. The fragmentation is especially severe in FIM because the tree is typically extremely wide (e.g. >25k vertices at level 4 for Gaz).

3.5.2 Scalability

Figure 3.4 shows the strong scaling results of traversal kernels. As expected, *DM* scales better compared to *SHM* for all the benchmarks except KC and LCP. This is because in *DM* runs, as more compute nodes are added, more hardware execution contexts become available and hence, the nodes are able to utilize all of the available

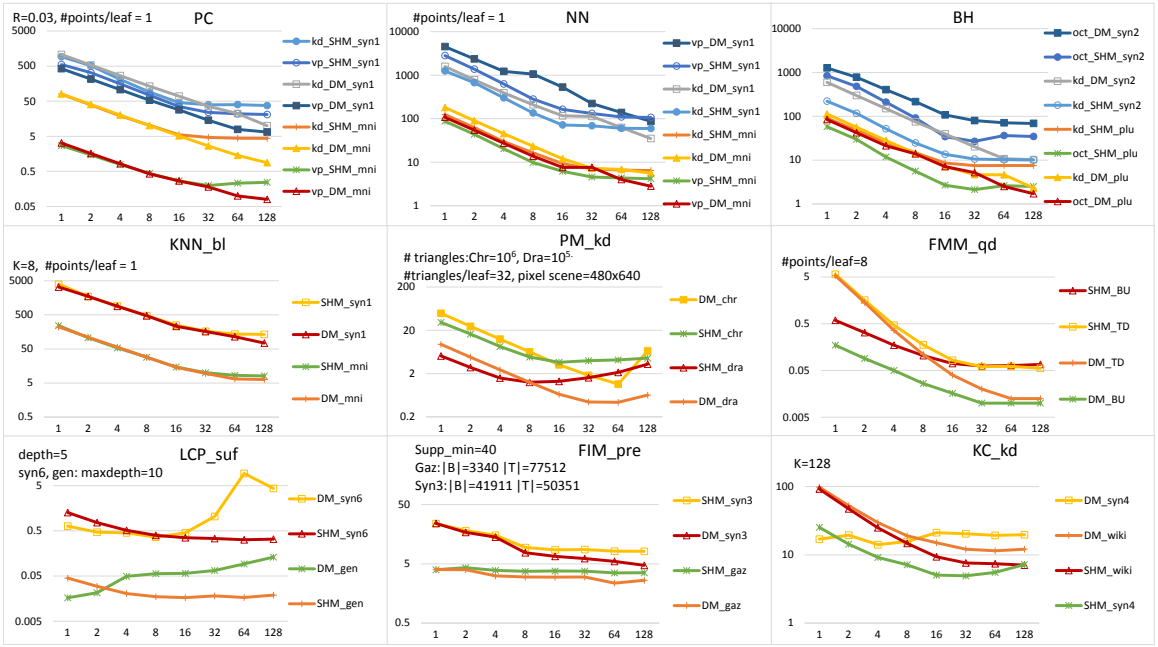


Fig. 3.4.: Scaling in Treelogy benchmarks. x-axis=Thread count (SHM)/Process count (DM), y-axis (log scale) =Runtime (s).

parallelism with minimal inter-node synchronization, thus yielding better scalability. However, scaling in *SHM* stops from 64 threads due to sharing of processing elements (cores) beyond 40 threads.

In case of KC, all the threads/processes synchronize at the end of every iteration to reconstruct the tree. Therefore, we see degraded performance of *SHM* versions beyond 32 processes (this behavior is observed in case of multiple-step runs of other iterative kernels such as BH, PM as well). Also, due to the increased overhead of inter-process synchronization, *DM* runs perform worse than *SHM* versions. Scaling is poor in case of LCP (both *SHM* and *DM*) because of low available parallelism and the extremely small traversal kernel: there is just a single post-order traversal to begin with and the child nodes of a vertex can be processed in any order before processing a parent vertex. When the traversal reaches a certain *depth*, specified as a tunable parameter, the child vertices are processed in parallel. We observe that even with the synthetic data of sufficiently long input strings, there are not enough child

vertices to be processed in parallel. Hence, the parallel overhead due to large number of threads/processes brings down the performance.

We also see a super-linear speedup for the top-down traversals of FMM. This is because of a great reduction in the lower-level cache accesses (e.g. on moving from 1 thread to 4 threads in *SHM* version, total L2 cache accesses reduced by 97.8% and L3 accesses reduced by 62.3% resulting in a speedup of 12.4x). As seen from Table 3.3, the top-down traversal step in FMM was nearly 9.5 times slower than the two bottom-up steps combined.

DM configurations mapped multiple processes to a single node of the cluster when a free node was not available. This, along with the data-parallel execution (entire tree replication on every process) is the cause for reduced performance at large numbers of processes and large input sizes. The *DM* run for PM with chr input (DM_chr) failed to execute with 128 processes due to memory limitations. Hence, we ran this configuration with the tree partitioned across all the nodes of the cluster and replicating only the top subtree (this is the pipelining strategy outlined by Hegde et al. [67]). As a result, due to the added communication overhead, we see the performance going down w.r.t. 64 process data-parallel run.

The GPU platform has the similar hardware structure as the *SHM*: all threads share the same limited hardware. Thus the scalability test result is also close to *SHM* (using real-world inputs). We assign each thread block 192 threads (6 warps) to take full use of a SMX and evaluate the scalability on the granularity of thread block. In Figure 3.5, before the number of thread blocks reaches 64[8], all benchmarks scale well and show a linear speedup as the blocks increase. For most benchmarks, scaling stops after 64 thread blocks, as the GPU resources are exhausted.

In summary, we find that these traversal kernels scale well, when taking advantage of ontology-driven optimizations. While the performance of BH with octree is better with clustered input, BH with kd-tree performs better with uniformly distributed data. However, BH with octree is known to perform better at larger scales [26]. We

[8]64 blocks contain 64*192 = 12,288 threads

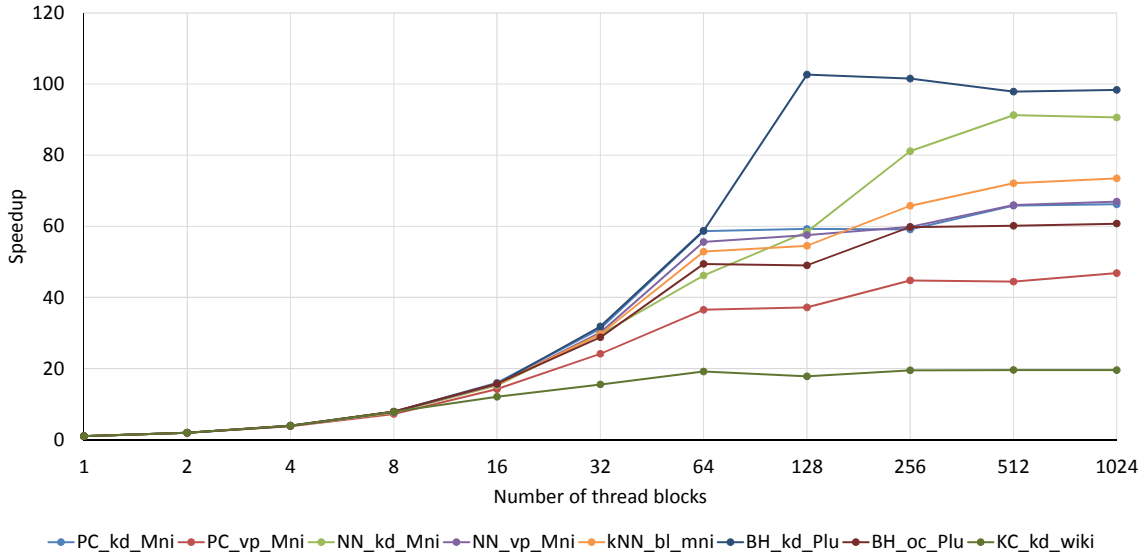


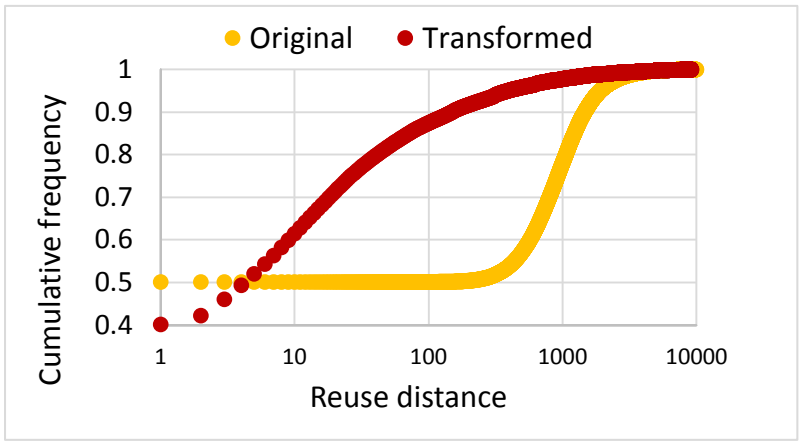
Fig. 3.5.: GPU scalability

also find that PC with vptrees attractive compared to PC with kd-trees because of vptrees facilitating efficient truncation, resulting in shorter average traversal lengths and hence better performance.

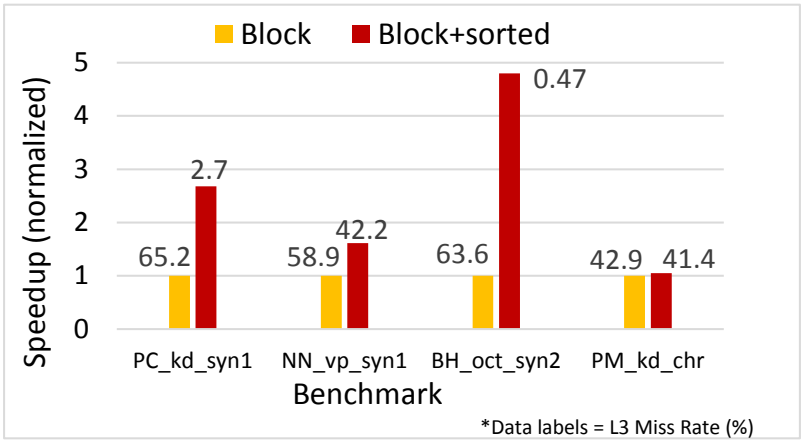
3.5.3 Case Studies

Figure 3.6a shows an example of using dual-trees and recursion twisting optimizations [68] to improve locality in PC with the help of our reuse distance analyzer tool. In this experiment, we evaluate PC_kd with a subset of mnist data set, since the average traversal length with mnist is the longest (see Table 3.3). The cumulative frequency of bigger valued reuse distances is higher for the transformed code than the original code. Hence, we can infer that the transformation has resulted in better temporal locality.

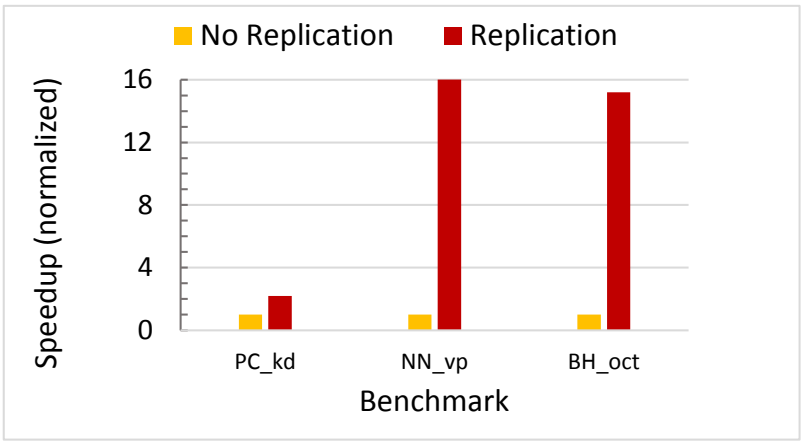
Figure 3.6b shows the effect of application-specific sorting optimization on temporal locality and the resulting improvement in performance in top-down kernels with independent traversals. These results have been published earlier by Jo et.al. [59] but are presented here for completeness. The *Block* configuration represents the *SHM*



(a)



(b)



(c)

Fig. 3.6.: Case studies: a) estimating locality benefits using reuse distance. b) improving locality through reordered traversal schedule. c) improving load-balance through subtree replication.

runtime values of Table 3.3. The *Block + sorted* configuration represents a reordered schedule of traversals in *Block* to obtain improved locality. These experiments are run with a block size of 4096. As expected, the performance improvement due to sorting is reflected in a commensurate decrease in cache miss rate. Note that *Block* and *sorted* are independent optimizations and can be applied to bottom-up kernels with independent traversals as well. In fact, *Block + sorted* is equivalent to the tiling optimization in FIM by Ghoting et.al. [56].

Figure 3.6c shows the effect of top subtree replication as a way of generalizing the LET optimization. These experiments are run with 128 processes and the tree is partitioned into subtrees of fixed height because the entire tree could be replicated. For these inputs, $|V|$ ranged from 80x10^6 to 256x10^6, and $|P|$ was 64x10^6. As expected, the top subtree presents a bottleneck when a large number of independent top-down traversals exist, and the node housing this subtree remains heavily loaded. Besides balancing the load, top subtree replication reduces communication overhead and hence, results in far better performance as seen from the figure.

3.6 Related Work

There are a wide variety of benchmark suites that focus on graph applications [49–51, 69], and several of these suites contain a handful of Treelogy benchmarks, such as Barnes-Hut, Frequent Itemset Mining, K-Means, and Fast Multipole. However, because these suites focus on a broader class of graph algorithms, they do not feature enough tree traversal algorithms to cover the wide variety of traversal structures that arise in such kernels. Graph-based, irregular application kernels have also been studied extensively in other works [52–54, 70]. However, analysis of tree based applications is not as extensive. Treelogy provides a way of characterizing the structural properties of tree algorithms, and ensures that its benchmarks span this space. Moreover, Treelogy provides reference implementations of these algorithms on multiple platforms, instead of focusing on single platforms as many prior suites do.

Irregular, tree-based applications have been mapped to massively parallel and distributed hardware platforms [24, 26, 48, 71, 72]. However, customized problem representations and transformations for applications have been proposed, analyzed, and optimized in a stand-alone context. While the parallelism profile of Barnes-Hut is presented in Lonestar [51], traversal properties are not studied. Performance of Nearest Neighbor search with multiple trees has been studied from a purely query response time perspective [45]. Treelogy differs from these works as it considers a broader class of tree traversal kernels, and multiple hardware platforms and tree type combinations. Treelogy can be used to inform generalizations of the many application-specific optimization strategies that prior implementations use.

3.7 Conclusions

In this work, we introduced an ontology for tree traversal kernels and presented a suite of tree traversals, Treelogy, with at least one kernel in each category. We evaluated multiple implementations of these traversal kernels using different types of trees and on multiple hardware platforms. We presented scalability analysis on different platforms, and an analysis of locality and load-balance with the help of metrics. We also presented case studies showing the effectiveness of certain optimizations. While the interest in tree based kernels comprising of traversals from various domains is increasing, we hope to expand the suite with more kernels and analyze their performance. Treelogy, by providing a wide variety of tree traversals, presents a useful target for developing and evaluating optimizations, and its ontology helps understand where and when those optimizations can be applied.

4. D2P: FROM RECURSIVE FORMULATIONS TO DISTRIBUTED-MEMORY CODES

4.1 Introduction

Programmers like recursive formulations of programs because they are straightforward to reason about and write, often have good locality properties, and readily expose parallelism [73, 74]. We want to create distributed parallel implementations *automatically* from simple shared-memory recursive specifications. We observe that the automation process is simplified when recursive formulations have certain properties: i) the data-dependencies of a recursive method are inclusive (i.e. the collective data-dependencies of all the recursive sub-invocations within the body of the method are a subset of the dependencies captured by method's arguments). ii) The data set intersection tests necessary for determining inter-task dependences are simplified.

Creating distributed-memory implementations involves several challenges that do not arise in shared-memory systems. First, we need to partition the data and computation into *tasks* of appropriate granularity split among different compute nodes so that different nodes can process portions of the data in parallel. Second, we need to insert communication between nodes to satisfy data dependences between tasks. Finally, we need to schedule tasks efficiently, to balance parallelism with the communication overhead. Automating the creation of distributed-memory programs requires that these challenges are automatically handled.

In recursive formulations with the inclusive property, each recursive "task" (think: invocation) touches a subset of the data of its parent task (think: caller method). Hence, coarse-grained tasks, suitable for a task-parallel model of computation, can be easily identified. When the tasks follow a specific order in computing the data sets, which either fully overlap or are disjoint, the data set intersection tests are greatly

simplified. Identifying that these properties in recursive divide-conquer formulations can simplify distributed-memory code generation is the main novelty of our work.

A specific set of programs that are important and satisfy these properties are recursive Dynamic Programming (DP) [75] programs. DP algorithms are very efficient in solving problems arising in domains such as bio-informatics, mobile communication, and finance. As some of these problems involve large data sizes that can exceed the memory capacity of a single compute node, distributed-memory solutions become necessary to process the entire data. Because formulating an efficient algorithm is a different challenge than performing data partitioning and task creation, communication insertion, and task scheduling, we would like a system where the concerns of *creating* an algorithm can be separated from the concerns of *distributing* it. In particular, we would like a system that automates as much of the distribution process as possible, allowing programmers to focus on simply designing algorithms.

While much of prior work has addressed the challenges of automating the distribution in different contexts for iterative codes of general programs, recursive programs are not explored as extensively. A majority of existing systems [76–80] work on iterative codes. In case of recursive programs, they adopt a profiling-based approach and assume a regular communication pattern [76]. Programmer created annotations in recent research exploits the inclusive property to optimize for locality and parallelism of recursive divide-conquer programs on shared-memory [73] and heterogeneous systems [8]. In case of DP programs, a majority use iterative codes. While most parallel implementations are for shared-memory systems, there exist only a few hand-tuned distributed-memory implementations [81–87]. Recent work has shown that efficient recursive formulations of DP algorithms outperform iterative codes on shared-memory systems primarily because of better cache utilization [88, 89]. Regarding distributing the compuation, recursive DP algorithms involve irregular communication patterns and offer a specialized (or simplified) domain. Hence, previous techniques for automation are not effective. Importantly, the inclusive and intersection properties are *implicit* in

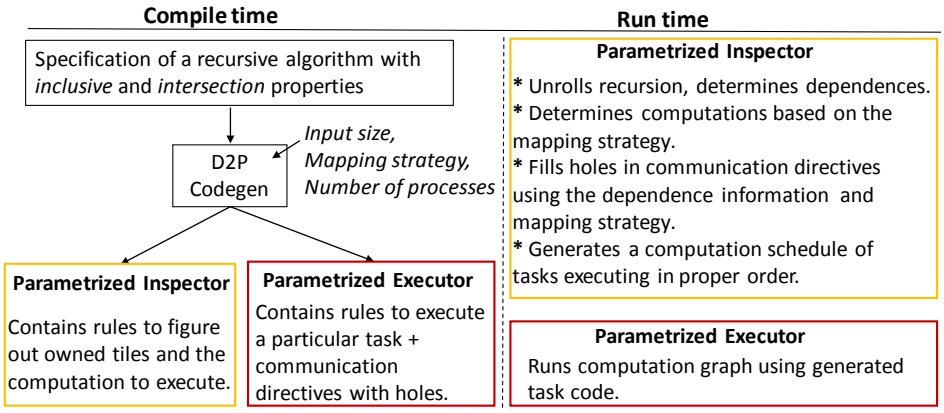


Fig. 4.1.: D2P system overview.

these recursive DP formulations and this is the key insight we exploit in *automatically* creating distributed-memory implementations of recursive DP algorithms.

4.1.1 Overview

In this paper, we present D2P, a framework that takes a shared-memory specification of a recursive algorithm with the inclusive and intersection properties (without any parallelism or partitioning specifications), and produces an MPI based, distributed-memory parallel code with recursive methods.

Figure 4.1 shows the overview. The input to D2P code generator is a specification, which is a high-level outline of a recursive algorithm. The specification need not mention the recursion base cases, which are application-specific. This attribute of the specification points to an important way in which D2P differs from other fully-automated code generation schemes [8, 76–78, 90]: user-assistance is necessary in filling the D2P-produced implementation with application-specific details to make it complete. Despite this semi-automatic nature, D2P completely automates (and hides) the challenging aspects of distributed-memory programming and delegates only the algorithm design aspects to an application designer.

At compile time, the D2P Codegen generates codes for the inspector and executor modules from the specification. Every process in a multi-process execution of the program executes these modules. The inspector and executor codes are parametrized on a data-partitioning (mapping) scheme, input size, and the number of processes, to enable the creation of an application code providing portable-performance. These parameters are provided at runtime. The inspector contains code for figuring out process-specific computations (method invocation with specific argument values) and the executor code contains rules (method bodies with communication directives) for performing those computations.

At runtime, the inspector module of every process determines tasks to compute, identifies task dependencies, establishes communication channels for each task, and prepares a schedule based on the identified task dependencies. The executor module of a process then executes the schedule.

First, the inspector *unrolls* a top-level recursive method multiple levels to create tasks. The unrolling hierarchically decomposes (breaks) the data into smaller parts and implicitly creates a tree. A vertex of the tree represents recursive method invocation computing a part resulting from the repeated decomposition up to the vertex's level. The children of this vertex are again method invocations called from within the body of the vertex's method. The internal vertices of the tree are *not involved* in computing the data: they only decompose the data into finer parts and in some cases do minimal work, such as initializing, before any of the children modify the data (due to the intersection property). Therefore, all the computation is done by leaf method invocations and hence, the leaves of the tree represent the tasks in D2P.

Based on a mapping scheme, the tasks are partitioned among processes in a deterministic manner known to all processes (facilitating determination of communication later). Task ownership follows data ownership as the processes follow an owner-computes rule (single owner for a data region) during computation. As method arguments are the only (entire) data read and written within the invocation (due to the inclusive property mentioned above), it is sufficient if the leaf method arguments

are inspected to determine inter-task dependencies. Unique IDs associated with data regions simplify this dependency computation. Finally, the inspector uses dependence information to insert communication—filling holes in communication directives, and identifying and establishing communication channels between processes to send/receive data as demanded by the task allocation.

At the end of inspector's execution, every process has a list of tasks. The executor begins executing a task from the list for which all the dependences are satisfied. As soon as a task completes execution, the executor sends the results of the computation to processes that own dependent tasks. Dependent task owners go through their respective task list to update task dependences and execute a task if the task dependences are fully satisfied. The result is a "single program multiple data" (SPMD) implementation employing asynchronous communication capable of running on a distributed-memory system.

The contributions of this paper are:

- We provide a framework D2P, which can take a specification of any recursive divide-conquer algorithm with the inclusive and intersection properties and convert the specification to a distributed-memory implementation capable of exploiting node- and core-level parallelism.

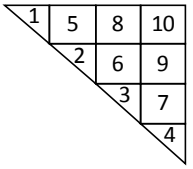
- We evaluate our framework on a set of recursive formulations having these properties: recursive DP algorithms. As an added feature, we augment D2P's code generator to parse specifications produced by an existing tool [74] that can automatically generate specifications for a *subset* of the recursive DP algorithms. Thereby, translating specifications of a wide class of DP problems to their respective MPI+Cilk-based implementations.

- We evaluate the scalability of D2P-generated implementations, preprocessing overheads, and compare against existing distributed-memory implementations.

Our evaluation shows that D2P implementations scale well and can admit extremely large problem sizes much beyond the memory capacity of a single node.

$$C(i,j) = \begin{cases} min(\ C(i,j),\ \min_{i<k<j} C(i,k) + C(k,j) + W(i,j,k)) & \\ 0 & j \leq\ i+1 \\ Given\ W(i,i+1,i+2), & where\ W(i,j,k)\ computes\ sum\ of\ Euclidian\ distances \\ & between\ (i,j),(j,k),and\ (k,i) \end{cases}$$

```
1  Cost(n){
2    table[n][n]; //n is number of vertices
3    for g ← 1 to n-1 do
4      for i ← 0 to n-g do
5        j ← i + g;
6        table[i][j] ← INFINITY;
7        for k ← i+1 to j–1 do
8          res ← table[i][k] + table[k][j] + W(i,j,k)
9          if res < table[i][j] then
10               table[i][j] ← res;
}
```



The recurrence equation specifies that only the cells of the upper-triangular matrix are computed. The numbered cells of an example 4x4 matrix are computed in the following order:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10

(a)

Fig. 4.2.: The Minimum Weight Triangulation problem: the recurrence equation computes the least cost of triangulating a convex polygon. This equation can be thought of as computing the cells of an upper triangular matrix. The figure shows the standard implementation scheme of computing the cells using an iterative code. Also shown is the order in which the code computes those cells.

We also observe that the overheads of task partitioning (task creation, assignment, and dependency determination) are negligible compared to the actual computation. Finally, we find that a D2P-generated program scales better than other distributed-memory implementations: the D2P-generated implementation of Smith-Waterman runs 59× to 412× faster than an implementation generated by an existing framework for parallelizing *iterative* DP formulations on distributed-memory systems. Also, the D2P-generated implementation outperforms a *hand-written*, application-specific implementation of Smith-Waterman in *most cases*, and is only 27% slower in the worst case.

4.2 Background and Motivation

Recursive algorithms with the inclusive and intersection properties, in particular, have advantages when it comes to distributed-memory implementations: every recursive method invocation accesses a data set, which is the super-set of the union of the data sets accessed within all recursive invocations done in the method body. In other words, the method arguments specify a bound on the entire data regions read/written, thus avoiding the need for programmer specified annotations. This also makes it easier to reason about data-dependencies necessary for automating parallelism extraction and data-partitioning. In comparison, dependency analysis is complicated in *iterative* codes with multiple layers of `for` loops with regular and irregular accesses. A recursive method invocation represents a coarse-grained computation and hence, a suitable target for the creation of a task. The recursive nature also exposes intra-task parallelism, which is naturally adapted by a task-parallel model of computation such as the Cilk [91] system. In recursive algorithms with the intersection property, a hierarchical decomposition (due to recursive calls) creates disjoint (unique) data partitions. Furthermore, the partitions are input-independent, computed as per a preorder execution of the recursion tree vertices, and can be represented by unique identifiers. As a result, inter-task dependences are computed easily.

Efficient recursive formulations—hereafter referred to as recursive formulations—of DP algorithms have additional advantages: i) control partitioning is not a concern due to the absence of branches in the formulation, ii) a static task partitioning scheme (due to a predictable amount of computation per task) can simplify communication insertion, and iii) simple, and unique identifiers (Z-order [92] numbers) can be assigned to data partitions resulting from hierarchical decomposition of the DP table. We next describe an example DP algorithm and its recursive DP formulation.

Minimum weight triangulation (MWT) [93] is commonly used in applications in finite element method and computational geometry, among others, to triangulate a convex polygon. The goal in MWT is to partition the polygon into triangles such

that the edges do not intersect and the sum of the edge lengths of the component triangles is minimized. A dynamic programming (DP) based approach to solving the MWT problem divides the problem into subproblems, computes solutions to the subproblems, and combines the optimal solutions of smaller subproblems to find an optimal solution for the whole problem. The recurrence equation in Figure 4.2 systematically computes the solution for the MWT problem. The intuition behind this equation is to partition the polygon into a simple triangle ($W(i, j, k)$) and sub-polygon(s) to the left and/or right ($C(i, k), C(k, j)$) of the simple triangle, compute the solutions for the sub-polygon(s) *separately*, then combine the solutions (*overlapping subproblems*). If the sub-polygons have an optimal solution, then the combined solution is *guaranteed* to be optimal (*optimal substructure*). The optimal substructure and overlapping subproblems properties are fundamental to problems that admit DP solutions [94].

The standard approach to implementing the recurrence equation is to use an "iterative" formulation. This consists of a set of nested `for` loops that iteratively compute the cells of a two-dimensional matrix (DP table) storing the solutions of all the subproblems computed. Figure 4.2(a) also shows the iterative code and its operation.

Every iteration of the `for` loop on line 4 can be run in parallel in this code. This parallelizing scheme reflects *wavefront-parallelism*, which is a popular approach for parallelizing iterative codes. The iterative code shown performs a bottom-up computation of finding optimal solutions to all subproblems of smaller sizes first before moving on to a bigger subproblem (computing all cells on the main diagonal before computing cells on the smaller adjacent diagonal). This computation order results in unpredictable runtimes, especially in the presence of cache sharing [74], and exhibits poor temporal locality compared to a recursive formulation of the MWT problem.

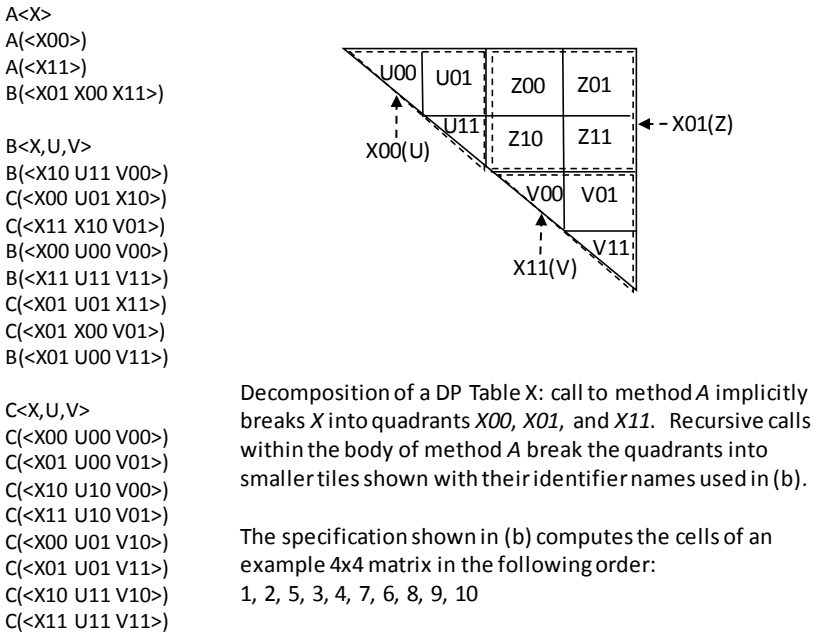


Fig. 4.3.: The specification: a) showing the outline of a recursive algorithm for the MWT problem. (b) shows the operation of this recursive formulation.

4.3 Design

D2P uses a specification of a recursive computation to generate a distributed implementation. We first describe what this specification looks like (Section 4.3.1). We then describe how D2P uses an inspector/executor approach to build a distributed implementation (Section 4.3.2). The inspector and executor are executed on each process in a SPMD manner. Finally, we describe some design details for how the inspector and executor work (Section 4.3.3).

4.3.1 Specification

The input to D2P is a specification, which is a high-level outline of a recursive algorithm. Figure 4.3 shows a specification, which is a recursive formulation for the MWT problem. The recursive formulation hierarchically breaks the DP table into smaller quadrants and computes their solutions. A D2P specification consists of a set of

recursive methods, among which there is only one top-level recursive method that calls other methods. We assume that the top-level recursive method is always specified first, followed by other methods. A method's definition starts with the method signature followed by the method body. Empty lines separate method definitions. Further, we assume that the first parameter of a method represents the data computed (written) and the remaining parameters represent the data read in order to compute the data. We also assume that the argument names in a recursive method call identify the data regions accessed (e.g. `X00` in `A(<X00>)` in the second line of Figure 4.3 identifies the *top-left* quadrant of a two-dimensional matrix `X`). The recursion base cases of the methods are omitted from the specification for the sake of simplicity.

The execution begins with a call to the top-level recursive method `A`, with the argument `X` representing the entire DP table. The call to method `A` implicitly breaks the problem into smaller subproblems (2x2 tiles of an example 4x4 matrix), represented as quadrants (`X00` - `X11`). The recursive calls to methods `A`, `B`, and `A` in the body of method `A` compute the first, second, and fourth quadrants (`X00`, `X01`, and `X11`) respectively. Note that the recurrence equation in Figure 4.2 specifies that only the upper triangular matrix is computed. Hence, only these three quadrants are computed. We expect the binary number used in the identifier names (`Xyy`, where `yy` is a two-digit binary number) to identify the quadrant and the DP table dimension (2D in MWT) as well.

Each of these recursive calls further breaks the quadrants into smaller parts (1x1 tiles identified by `Uyy`,`Vyy`, and `Zyy`). For example, the call to `B(X01,X00,X11)` in method `A`'s body (line 4) computes the second quadrant `X01` by reading quadrants `X00` and `X10`. To compute `X01`, method `B` first breaks `X01` (and `X00`, `X10` as well) into smaller quadrants. The first call within method `B`'s body, `B(Z10,U11,V00)`, illustrates this: it computes the third quadrant `Z10` of `Z`, which is the same as `X01`, by reading the fourth quadrant `U11` of `U`, and the first quadrant `V00` of `V` (`U` and `V` are the same as `X00` and `X11` resp.). The recursion stops here because it is not possible to decompose the 1x1 tiles (cell) further.

The base case defines how a cell or a tile (group of cells) is computed from other cells (lines [8-10] in `Cost` method of Figure 4.2(a)). Method signatures of `B` and `C` in the specification tell that these methods compute a cell (tile) by reading two cells (tiles), `U` and `V`. Hence, the base case is same for both these methods. Method `A` computes a cell from initial conditions (`W(i,i+1,i+2)`) and hence, has no dependencies. Therefore, `A`'s base case is different from methods `B` and `C`. By default, the base case computes a single cell in D2P. Note that in addition to the base case, the specification omits: i) input size necessary to identify the exact data region within the 2D matrix (e.g. a 2x2 square tile containing cells from $(0, 2)$ to $(1, 3)$—`X01` and ii) the data-type of the cell (`CELLTYPE`—`float/double` for the MWT problem). The end-user is expected to insert code corresponding to these in the D2P generated code (see Section 4.4, auxiliary methods, for details.). As the specification does not contain any DP-specific properties, D2P could generate code for any recursive divide-conquer algorithm with the inclusive and intersection properties. Next, we explain how D2P converts a specification to a distributed-memory parallel code.

4.3.2 Inspector and Executor

A task in D2P computes a method call with specific argument values. Therefore, a task constitutes: i) a pointer to the method, ii) argument values (iii) input, and (iv) output dependences. The input dependences are a set of task IDs and the output dependences are a set of process IDs. The job of determining what the tasks should be and what the dependences are is delegated to an inspector that D2P generates. Executing the tasks while respecting these dependences is delegated to an executor that D2P generates. We next describe at a high level how these inspectors and executors are generated for each process. Details about how the inspector determines which tasks a process owns, and what the task's dependences are, are described in Section 4.3.3.

Listing 4.1: Operation of the D2P inspector generator

insert code to:

I. Generate a map M of data region IDs to task IDs: data region ID is the ID of the write parameter, and the task ID is that of the last task that wrote the region.

II. **for** each method m A(X, Y, Z) defined in spec:

1. Generate inspector method signature
   a. Produce full types for each parameter.
   b. Each parameter also has metadata parameters of bounding-box and ID of the data region that a parameter represents.
   c. Add stack depth parameter to track unroll depth.
2. Generate inspector base case
   a. Base case triggers when stack depth = unroll depth.
   b. Generate unique task ID.
   c. Insert region ID (X's ID) and task ID into map M.
   d. Based on X's ID, decide which process owns a task writing to X
      i) If self (i.e. process P_{self})
      -Create a task T.
      -Figure out input and output dependences for task T.
      input deps:
      -Use map M to find T_{lastID}, ID of the last task that wrote X.
      -Add T_{lastID} to T's input deps.
      -Use the list of tasks to get T_{last}, the task with ID T_{lastID}.
      -Add T's owner process ID(i.e. P_{self}) to T_{last}'s output deps.
      / tasks are created in execution order. So, this creates correct input dependences. For the same reason, T's output dependences are not known yet. They will be figured as explained in the next step */*
      -Add T to P_{self}'s list of tasks
   e. **for** each read parameter R: {Y, Z}
      i) If R is not the same as write parameter X:
      -Use map M to find $T_{lastRID}$,ID of the last task that wrote R.
      -Process P_{self} adds $T_{lastRID}$ to T's input deps
      -Based on R's ID, decide which process owns T_{lastR},

```
        the last task that wrote R. Let P_ownsR be the owner.
      -Process P_ownsR adds P_self to T_lastR's output deps.
  3. Generate recursive case
    a. Create sub regions (e.g. X00, Y01, Z11), their bounding-boxes,
     and compute their region IDs.
    b. Call inspector methods as per the spec (use arguments created
     in step a).

III. Generate wrapper that calls the top-level inspector method:
  1. Create arguments to call the top-level inspector method.
   -Bounding-box for the data region (whole matrix): this depends
    on input size(size). For a 2-dimensional grid, cells(0,0) to
    (size-1, size-1) are computed.
   -data region: nullptr
   -data region ID: 0
   -call stack depth: 0
   /*Since the computation is distributed among tasks, memory is
   not allocated to the whole matrix (data region is nullptr).
   As tasks compute sub-data regions, memory for a sub-data region
   is allocated when the the first task that writes to that
   sub-data region is created*/
  2. Call top-level inspector method with arguments created.
```

Listing 4.2: Inspector generator input (definition of method B from the spec), and output code produced.

```
/* Input */
B<X,Y,Z>
B(<X10 Y11 Z00>)
C(<X00 Y01 X10>)
C(<X11 X10 Z01>)
B(<X00 Y00 Z00>)
B(<X11 Y11 Z11>)
C(<X01 Y01 X11>)
C(<X01 X00 Z01>)
B(<X01 Y00 Z11>)
```

```
/*Output code generated by the inspector generator.
The following globals are generated only once. rank and unrollDepth
are the other globals used. rank is the process rank set by MPI
environment. unrollDepth is set based on number of processes used
in execution.*/
map<int, int> M;
map<int,Task> L;
int taskID=0;

/*A method definition corresponding to the method def from the spec:
X Y and Z are data regions. IDx, IDy, IDz are region identifiers
of X, Y, and Z resp. They are z-order numbers. bX, bY, and bZ are
bounding boxes of the data regions X, Y, and Z resp. They store the
top-left and bottom-right coordinates. The type Box is defined in
common.h. It is defined based on the dimension extracted from the
input (in this example DIMENSION 2). */
void B_unroll(CELLTYPE* X, Box* bX, int IDx, CELLTYPE* Y, Box* bY,
      int IDy, CELLTYPE* Z, Box* bZ, int IDz, int callStackDepth) {

    if(callStackDepth = unrollDepth) {
      if(GetOwner(IDx) == rank) {
        /*Create task. FuncPtrB is the function pointer type of the
        executor method named B--defined in the next section.
        The function pointer 'B' and the exact argument values of X,
        bX, Y, bY, Z, bZ, and callStackDepth are stored as C++ tuple*/
        Task<FuncPtrB, CELLTYPE*, Box*, CELLTYPE*, Box*, CELLTYPE*,
            Box*, int> T(B, X, bX, Y, bY, Z, bZ, callStackDepth);

        /*compute input dependences for T: if the task list M
        contains a task with ID IDx (this is true when some task has
        already computed tile X before. In this case, the current task
        must read and update tile X rather than overwriting.) then
        add that task's ID to T's input deps. Also, update T_last's
        output deps to send data to T.*/
```

```
      if(M.exists(IDx)) {
          T.inDeps.add(M[IDx]);
          L[M[IDx]].outDeps.add(rank);
      }
      //add T to list of tasks
      L[taskID]=T;
    } //end if(GetOwner..

    //update the map with the ID of the latest task writing X.
    M[IDx] = taskID++;
    //Inspect read parameters and compute input and output deps
    int tmpIDs[2]={IDy, IDz};
    for(int i=0;i<2;i++) {
        if(tmpIDs[i] != IDx) {
            if(GetOwner(IDx) == rank)
                T.inDeps.add(M[tmpIDs[i]]);
            if(GetOwner(tmpIDs[i] == rank))
                L[M[tmpIDs[i]].outDeps.add(GetOwner(IDx));
    }
    return;
}

/*Recursive case: first define identifiers used. Bounding boxes of
quadrants can always be computed from bounding box of the parent.*/
Box X00(X->coords[0],X->coords[1],X->coords[2]-(X->coords[2]-
 X->coords[0]+1)/2,X->coords[3]-(X->coords[3]-X->coords[1]+1)/2);
Box X01(X->coords[0]+(X->coords[2]-X->coords[0]+1)/2,X->coords[1],
 X->coords[2],X->coords[3]-(X->coords[3]-X->coords[1]+1)/2);
Box X10(X->coords[0],X->coords[1]+(X->coords[3]-X->coords[1]+1)/2,
 X->coords[2]-(X->coords[2]-X->coords[0]+1)/2,X->coords[3]);
Box X11(X->coords[0]+(X->coords[2]-X->coords[0]+1)/2,X->coords[1]+
  (X->coords[3]-X->coords[1]+1)/2,X->coords[2],X->coords[3]);
... //similarly Y and Z quadrants are defined.

B_unroll(X, &X10, IDx*4+2, Y, &Y11,
```

```
      IDy*4+3, Z, &Z00, IDz*4+0, callStackDepth+1);
    C_unroll(X, &X00, IDx*4+0, Y, &Y01,
      IDy*4+1, X, &X10, IDx*4+2, callStackDepth+1);
    C_unroll(X, &X11, IDx*4+3, X, &X10,
      IDx*4+2, Z, &Z01, IDz*4+1, callStackDepth+1);
    B_unroll(X, &X00, IDx*4+0, Y, &Y00,
      IDy*4+0, Z, &Z00, IDz*4+0, callStackDepth+1);
    B_unroll(X, &X11, IDx*4+3, Y, &Y11,
      IDy*4+3, Z, &Z11, IDz*4+3, callStackDepth+1);
    C_unroll(X, &X01, IDx*4+1, Y, &Y01,
      IDy*4+1, X, &X01, IDx*4+3, callStackDepth+1);
    C_unroll(X, &X01, IDx*4+1, X, &X00,
      IDx*4+0, Z, &Z01, IDz*4+1, callStackDepth+1);
    B_unroll(X, &X01, IDx*4+1, Y, &Y00,
      IDy*4+0, Z, &Z11, IDz*4+3, callStackDepth+1);
    return;
}

/*Wrapper to call top-level inspector method*/
void Unroll() {
  /* inputSize[i] denotes the number of cells in dimension i
  This is initialized by the Auxiliary function ReadInput.
  The top-level inspector method will be called A_Unroll (
  if the top-level executor method is called A. In this example,
  A accepts just 1 parameter (X) as read from the spec.*/
  Box b(0,0,inputSize[0]-1, inputSize[1]-1);
  A_Unroll(nullptr, &b, 0, 0);
}
```

The goal of the inspector, when executed, is to produce a list of tasks, where each task computes a data region based on reading other regions, as shown in Listing 4.1. The inspector does so by constructing a set of mutually-recursive functions, corresponding to the functions in the specification. These functions execute at runtime to generate the list of tasks. When the recursion reaches a leaf, this corresponds

to a task that must be executed. If this task is owned by the current process, the inspector computes the dependences for this task and places it in the process's task list (Section 4.3.3 describes how task ownership is determined and how dependences are computed). When the inspector module of each process completes execution, all the task dependencies are computed and each process gets a per-process list of tasks. Note that because the inspector generates tasks only from leaves of this recursive structure, it is essentially *unrolling* the recursion of the specification to identify leaf tasks. Input and output of inspector generator is shown in Listing 4.2.

Listing 4.3: Operation of the D2P executor generator

```
insert code to:
 I. for each method m A(X, Y, Z) defined in spec:
  1. Same as II.1 in Listing 4.1 except each parameter has one
     metadata parameter of bounding-box.
  2. Same as II.2 in Listing 4.1 except base case triggers when
     bounding-box size=1. Also the base case is empty and needs to be
     filled-in by the end-user.
  3. Same as II.3 in Listing 4.1 except generate two versions of the
     code(II.3.b) corresponding to serial and parallel execution.
     a. parallel version triggers when call stack depth < SPAWNCUTOFF.
        -Insert cilk_spawn and cilk_sync appropriately for calls that
        can be executed in parallel.
        /*parallelism is specified through annotations now*/
     b. Serial version triggers when call stack depth >= SPAWNCUTOFF.

 II. Generate wrapper that listens for incoming messages:
  1. Scan the task list for a task with all dependences satisfied.
  2. if there exists such a task T
     a. Execute task T.
     b. When done, write T's ID in the metadata of the data
        region that T computed. Send data region + metadata to
        processes mentioned in T's output deps.
     c. Remove T from task list
  3. If the task list is empty stop listening.
```

```
3. Otherwise, block on (listen to) incoming messages
 -Triggers when there is an incoming message
 a. Retrieve producer task ID T_prd from metadata in the message
 b. For each task T in list:
  -If T_prd is in T's input deps:
    -Remove T_prd from input deps.
    -Set T's argument value to message received.
 goto step II.1
```

Listing 4.4: Executor generator input and output code produced

```
/*An example input (method definition with parallelism specification).
Here we assume that parallelism specification is provided in the
input spec. If the parallel annotations are absent, then D2P executor
generator does not extract parallelism currently. As a result, all the
calls to recursive methods within a method body would be executed serially.
Annotations to the inspector generator input makes no difference.*/
B<X,Y,Z>
B(<X10 Y11 Z00>)
parallel: C(<X00 Y01 X10>) C(<X11 X10 Z01>)
parallel: B(<X00 Y00 Z00>) B(<X11 Y11 Z11>)
C(<X01 Y01 X11>)
C(<X01 X00 Z01>)
B(<X01 Y00 Z11>)

/* Output code produced in RecursiveMethods.cpp */
void B(CELLTYPE* X, Box* bX, CELLTYPE* Y, Box* bY,
        CELLTYPE* Z, Box* bZ, int callStackDepth) {

  if((X->coords[0]==X->coords[2]) && (X->coords[1]==X->coords[3])){
    /* User fills in application-specific code here. For MWT, this
    would be:*/
    int i=X->coords[1]; int j=X->coords[0]; int k=Y->coords[0];
    if(i == j) return;
    CELLTYPE w_ikj=Weight(i,j,k); //Weight(i,j,k) given as input.
    /*GetDPTableCell(i,j,X) fetches the memory location within
```

```
  a data region X pointed to by coordinates i,j. Because (i,j)
  are absolute coordinates (e.g. <3,15>), to compute the
  offsets correctly within a tile, metadata information of
  top-left and bottom-right coordinates is stored along with
  the tile*/
  CELLTYPE* cost_ij = GetDPTableCell(i,j,X);
  CELLTYPE* cost_ik = GetDPTableCell(i,k,Y);
  CELLTYPE* cost_kj = GetDPTableCell(k,j,Z);
  CELLTYPE newCost = *cost_ik + *cost_kj + w_ikj;
  if(newCost < *cost_ij)
      *cost_ij =  newCost;
  return;
}
/*bounding boxes of quadrants can always be computed from
the parent bounding box. Same code as in the inspector's
method output shown previously*/
/* SPAWNCUTOFF by default is set to unrollDepth+2
if(callStackDepth < SPAWNCUTOFF) {
  B(X, &X10, Y, &Y11, Z, &Z00, callStackDepth+1);
  C(X, &X00, Y, &Y01, X, &X10, callStackDepth+1);
  cilk_spawn
  C(X, &X11, X, &X10, Z, &Z01, callStackDepth+1);
  cilk_sync;
  B(X, &X00, Y, &Y00, Z, &Z00, callStackDepth+1);
  cilk_spawn
  B(X, &X11, Y, &Y11, Z, &Z11, callStackDepth+1);
  cilk_sync;
  C(X, &X01, Y, &Y01, X, &X01, callStackDepth+1);
  C(X, &X01, X, &X00, Z, &Z01, callStackDepth+1);
  B(X, &X01, Y, &Y00, Z, &Z11, callStackDepth+1);
}
else {
  B(X, &X10, Y, &Y11, Z, &Z00, callStackDepth+1);
  C(X, &X00, Y, &Y01, X, &X10, callStackDepth+1);
  C(X, &X11, X, &X10, Z, &Z01, callStackDepth+1);
```

```
    B(X, &X00, Y, &Y00, Z, &Z00, callStackDepth+1);
    B(X, &X11, Y, &Y11, Z, &Z11, callStackDepth+1);
    C(X, &X01, Y, &Y01, X, &X01, callStackDepth+1);
    C(X, &X01, X, &X00, Z, &Z01, callStackDepth+1);
    B(X, &X01, Y, &Y00, Z, &Z11, callStackDepth+1);
  }
  return;
}


/*Wrapper to execute Tasks in L produced by the inspector*/
void ExecuteTasks() {
  while(true) {
   for(int i=0;i<L.size();i++) {
    /*execute a task if the input dependences are satisfied*/
    if(L[i].inDeps.size() == 0) {
      /*check the function pointer type saved in a task */
      switch(type(L[i].funcPtr)) {
      ...
      /* if the pointer points to method B*/
      case B:
      {
          /* Here param[i] are the argument values of X, bX,
          Y, bY, Z, bZ, and callStackDepth saved in the
          inspector's method B_Unroll */
          ((B*)(L[i].funcPtr))(L[i].param[1], L[i].param[2],
          L[i].param[3], L[i].param[4], L[i].param[5],
          L[i].param[6], L[i].param[7]);
      }
      break;
      ...
    }
    /*param[1] is write parameter (param[0] is function ptr)
    the first entry in a tile is the task ID.*/
    L[i].param[1][0] = L[i].ID;
```

```
      for(int j=0;j<L[i].outDeps.size();i++)
        SendResults(L[i].param[1], L[i].outDeps[j]);
      L.delete(L[i].ID);
     }
     //if there are no tasks to execute exit the loop.
     if(L.size() == 0)
      break;

     //wait for incoming messages (block on a wildcard message)
     CELLTYPE* msg = WaitForIncomingMsg();
     int inTaskID = msg[0];
     for(int i=0;i<L.size();i++) {
      if(L[i].inDeps.exists(inTaskID)) {
      /*the incoming message is the argument value. Plug it at
      the right place. If msg received is tile X11 in a task
      waiting to execute call A(X01, X00, X11), then msg must
      be plugged as param[5] accounting for other metadata
      params (param[1] is X01, param[2] is bX01 (boxX01),
      param[3] is X00, param[4] is bX00, param[5] is X11, param[6]
      is bX11, param[7] is call-stackDepth*/
         L[i].params[inDeps[inTaskID]] = msg;
         //update input dependencies
         L[i].inDeps.remove(inTasskID);
      }
     }
   }
}

/* common.h */
#define DIMENSION 2
#define METADATASPACE (2*DIMENSION+1)
typedef int CELLTYPE;
inline CELLTYPE* GetDPTableCell(int i, int j, CELLTYPE* data) {
 CELLTYPE* cell = data+METADATASPACE;
 int side = data[DIMENSION];
```

```
 int iOffset=i - data[1];
 cell += (iOffset*1* side);
 int jOffset=j - data[0];
 cell += jOffset;
 return cell;
}

typedef struct Box
{
        int coords[2*DIMENSION];
        Box(){}
        Box(int a, int b, int c, int d) {
          coords[0]=a;coords[1]=b;coords[2]=c;coords[3]=d;
        }
        Box(const Box&b) {
          coords[0]=b.coords[0];coords[1]=b.coords[1];
          coords[2]=b.coords[2];coords[3]=b.coords[3];
        }
        bool operator==(const Box& rhs) {
          bool flag = true;
          if((this->coords[0]!=rhs.coords[0]) ||
                (this->coords[1]!=rhs.coords[1]) ||
                (this->coords[2]!=rhs.coords[2]) ||
                (this->coords[3]!=rhs.coords[3]))
                  flag=false;
         return flag;
        }
        long int GetBoxSize() {
          long int len = 1;
          for(int i=0;i<DIMENSION;i++) {
            len *= (coords[i+DIMENSION]-coords[i]+1);
          }
          return len;
        }
}Box;
```

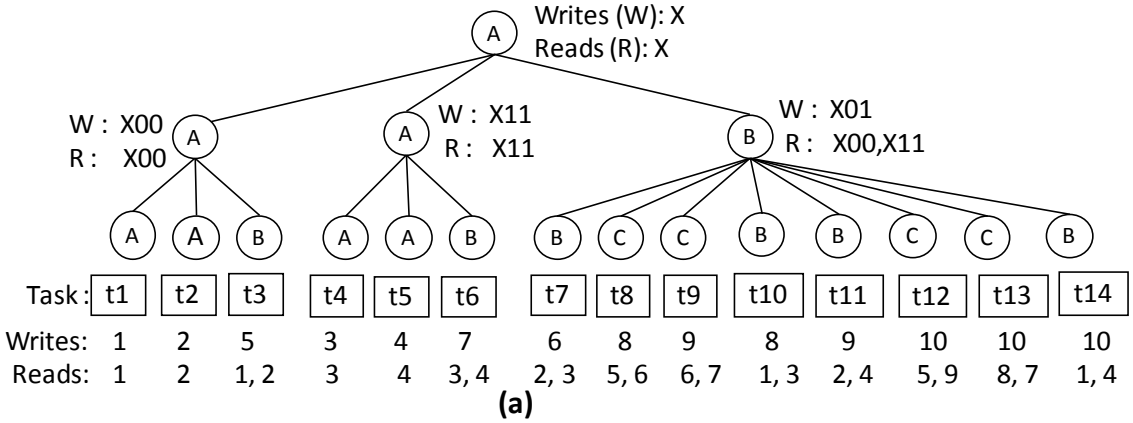
The goal of the executor is to execute the list of tasks that the inspector produced. Figure 4.3 shows how the skeleton for that executor is constructed. The executor module of each process spins in a loop scanning the task list produced by the inspector. A task is executed when either all its input dependences are satisfied or the set of input dependences is empty. When a task completes execution, it is removed form the list and the executor immediately sends the results of the computation (a data region / tile in a matrix) to all the processes specified in the output dependences. Before sending the results, the executor attaches additional metadata that indicates the task ID that computed the results and the bounding box information of the tile computed. The executor waits (blocks on a wildcard message) when there are no tasks to execute.

Upon receiving a message, the executor updates the input dependencies for all tasks waiting on the task mentioned in the metadata received. The executor then executes any task for which all the dependences are satisfied. The executor exits the loop when the list of tasks is empty. Listing 4.4 shows the input and output of the executor generator.

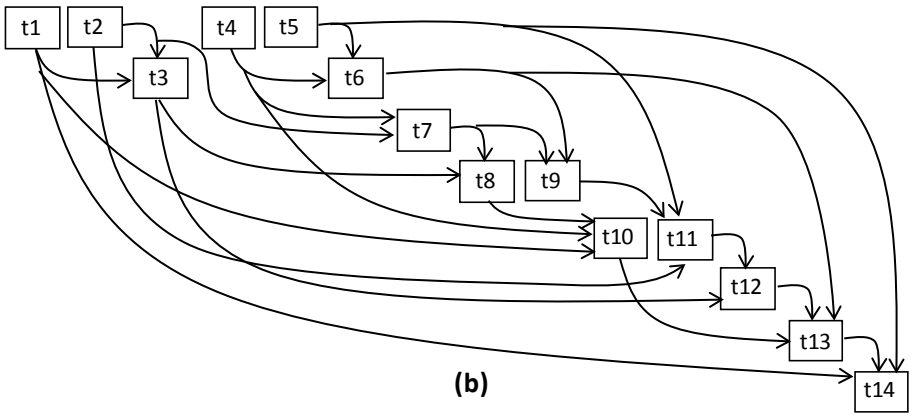
4.3.3 Design Details

We next explain inspector's operation at runtime. Figure 4.4 illustrates task creation, dependency inference, and task partitioning.

Unrolling depth Recall that the inspector unrolls the computation to a certain depth until it arrives at base cases that it turns into tasks for the executor. With a level of unrolling, the DP table is broken into smaller parts, so more unrolling leads to smaller, but more numerous tasks. Deciding the optimal granularity of these tasks—i.e., number of levels of recursion to unroll—is hard. Typically, when executing a recursive parallel program on a single compute node, the tradeoff is between the overhead of creating new tasks and the amount of parallelism exposed. For a distributed-memory implementation, the number of levels of recursion to unroll, D, reflects a tradeoff between communication overhead and the available parallelism: a larger D introduces

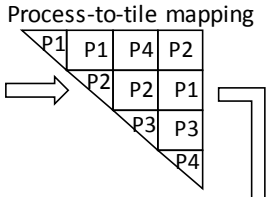


(a)



(b)

Tasks are partitioned among 4 processes based on a mapping scheme of assigning the tiles along the diagonals to processes in round-robin order.



Process	P1	P2	P3	P4
Tiles owned	1, 5, 9	2, 6, 10	3,7	4,8
Task owned	t1,t3,t9,t11	t2,t7,t12,t13,t14	t4,t6	t5,t8,t10

(c)

Fig. 4.4.: D2P inspector in action: (a) shows task creation by unrolling MWT's top-level recursive method A two levels deep. The leaves of the recursion tree are identified as tasks. The Figure also shows the numbers of tiles read and written by each task (numbering follows from Figure 4.2). (b) shows inter-task dependences. (c) shows task partitioning among 4 processes.

more tasks, which may increase the available parallelism but also introduces more data dependences, and hence communication. Figure 4.4(a) shows the tasks created when D=2.

Identifying data dependences via effect intersection Figure 4.4(b) shows the data dependences among tasks resulting from unrolling. Correct dependences can be derived after computing a linearization of the tree in preorder traversal (to ensure that dependences are resolved in program order). The dependency structure, as the arrows indicate, can be computed using simple set intersection (of the tile numbers) given the tile numbering information. The source of the arrow is a task writing to a DP table tile (producer) and the destination is a task that reads these tiles (consumer). Enforcing these data dependences is all that is necessary to ensure correct execution, due to the inclusion property of the dependences as described earlier.

Note, interestingly, that because D2P performs unrolling of the original recursive algorithm, it is able to expose *more* parallelism than the original formulation. This is because the original formulation uses control dependences (in the form of sequential ordering) to conservatively enforce data dependences that arise between coarse-grained tasks. Two coarse-grained tasks x and y may have to execute sequentially because some subtask in y is dependent on some subtask in x, even though *other* subtasks in y can execute in parallel with x. As a concrete example, note that in the original recursive formulation of MWT in Figure 4.2(c), the first `B` task executes sequentially after the parallel execution of the first two `A` tasks. However, in the unrolled dependence graph of Figure 4.4(b), we see that this is conservative: t_7 (arising from a `B` task) can execute in parallel with t_3 and t_6 (arising from `A` tasks) despite the control dependence in the original formulation. D2P enforces exactly these data dependences, and hence is able to expose more parallelism than the original parallel recursive implementation, depending on how much unrolling is done. Of course, performing additional unrolling can incur additional runtime overhead and communication, so this tradeoff must be carefully managed.

Task assignment D2P uses an *owner-computes* strategy: the inspector assigns DP table tiles to processes as per the mapping strategy provided at runtime, and then *all* tasks that compute the tiles mapped to a process are assigned to that process. This ensures that the task-to-process assignment results in a single owner for a tile. Note that this simplifies communication, as we do not need to consider different processes writing to the same tile. D2P provides knobs to programmers to control the mapping strategy (see Section 4.4). Figure 4.4(c) shows an example mapping strategy of assigning tiles along the diagonals to four processes in round-robin order. The Figure also shows the resulting tile and task distributions.

Communication insertion and data movement In a distributed-memory setting, if the consumed data does not reside on local memory, then the producer and consumer of the data must establish communication channels and perform communication. For a task in D2P, producers are the *tasks* mentioned in its input dependences. However, the consumers are the *processes* mentioned in its output dependences. These dependences are computed during inspector's execution (Listing 4.1, II.2.d and III.2.e). This computation is dependent on the ability of processes to figure out the owner of a tile (data region) given a tile number and a mapping scheme. E.g. in Figure 4.4(a) task t_{12} writes to tile 10 and reads from tiles 5 and 9. when t_{12} is being created, last tasks to have computed tile 5 and tile 9 are t_3 and t_{11} respectively. The last task information is available to process $P2$ (owner of tile 10, and hence, task t_{12})—and all the processes—due to the SPMD code updating map M (Listing 4.1, II.2.c). Therefore, process $P2$'s inspector updates task t_{12}'s input dependences to expect data from tasks t_3 and t_{11}. The inspector of process $P1$ (owner of tiles 5 and 9) updates the output dependences of tasks t_3 and t_{11} to send their results to $P2$.

Once the inspector establishes communication channels between producers and consumers, the actual data movement follows during executor's execution of the tasks. Since in D2P, the producers of the data push the data to consumers, consumer process IDs are essential for the tasks to correctly perform communication. A consumer

process delivers the data received for consumption to its task(s) based on the input dependences of the task(s). The data is delivered to the correct task if its input dependences contain a task ID that matches the task ID mentioned in the metadata received along with the data (Figure 4.3, II.3 and II.2.b).

Leaf-task implementation One advantage of D2P is that end users can rely on D2P for orchestrating inter-task parallelism as shown in Figure 4.4 while continuing to use their highly-optimized, single-process code within a task. Because a task computes a recursive method with inclusive dependences, users can replace the method body with codes offloading computation to multiple threads, vectorizing processors, GPUs etc. This decoupling of a task's implementation is safe because of the following: if sub-parts (data computed by an optimized iterative code or spawned sub-tasks) of two coarse-grain tasks overlap, then the inputs to the coarse-grain tasks must overlap because of the inclusion property. As a result, there would be a dependency between the coarse-grain tasks. And since the coarse-grain tasks execute atomically, we are guaranteed to wait for all dependences to be satisfied. Thus, *no two tasks that could interfere can possibly execute in parallel.*

4.4 Implementation

Intra-task parallelism While D2P primarily concerns itself with identifying and exploiting inter-task parallelism, as shown in Figure 4.4(b), each task itself represents a fairly coarse-grained computation. D2P leverages the resulting *intra*-task parallelism—easily exposed due to the recursive nature of the implementation—by using Cilk `spawn` [95] to implement parallel sections within a leaf-method body (though, as mentioned in Section 4.3, the user can instead replace a leaf-method recursive implementation with their own optimized single-process code).

While executing a recursive implementation of a leaf-method, spawning sub-tasks down till the terminating case can greatly increase the parallel overhead. The D2P Codegen generates two versions of the method body (with and without Cilk keywords)

and the executor executes the sequential version (without Cilk keywords) to control this overhead. The executor stops executing the parallel version beyond C levels of recursion. We set the spawn-cutoff C to two by default, since, this created sufficient number of sub-tasks for all the benchmarks and the input sizes considered.

Cilk workers (pthreads analogue) execute the spawned sub-tasks. The number of Cilk workers per process reflect a tradeoff between shared-memory parallelism vs. communication-computation overlap: we have two options to subscribe to a fixed number of parallel hardware contexts (cores)—(1) use single Cilk worker per process and more processes or (2) multiple Cilk workers per process and lesser number of processes. With option 1, there is more fine-grained parallelism since the inspector unrolls more with more processes. The increased fine-grained parallelism provides more opportunities to overlap increased communication with parallel computation. With option 2, lesser processes enforce shallower unrolling, which makes a task compute a larger tile. Larger tiles provide more opportunities to map computation to multiple Cilk workers more effectively. However, as the example on effect intersection discussed in Section 4.3 showed, increased task granularity imposes artificial constraints on independent sub-tasks that belong to distinct tasks (Cilk workers in D2P do not communicate with workers from other tasks). Hence, there are lesser opportunities for overlapping communication with computation. This may be acceptable for compute-heavy applications, where communication is not a significant factor. However, for memory-bound applications, we prefer having more fine-grained parallelism. We set the number of Cilk workers to 1 by default and provide a knob to set the Cilk workers. Cilk workers in D2P are not involved in data exchange since the granularity of data communicated is a tile and Cilk workers compute sub-tiles. The process owning a leaf-task sends a tile of data to dependents only when all the Cilk workers executing sub-tasks of the leaf-task finish executing.

Parameters - inspector We provide blocked, block-cyclic, and hybrid strategies for mapping of data regions to processes. In case of many DP problems, we observe

that the data dependences exist along the rows and columns of the DP table, and parallelism is along the diagonal (MWT) or anti-diagonal. Hence, we decompose the problem until the number of tiles in a column (or row) are equal to the number of processes. This avoids column (or row)-wise communication, while still assigning tasks computing tiles along diagonals and anti-diagonals to different processes. As a result, this strategy reduces communication while maintaining parallelism for many DP problems. This strategy also defines the base case trigger in the inspector (Listing 4.1, II.2.a): the unroll depth D, which hierarchically decomposes a problem (hence, creating the tiles), is dependent on the number of processes. The tiles can only be created when input sizes are known (Listing 4.1, III.1-size). Hence, the mapping scheme, number of processes, and the input size are the required parameters for the inspector.

Data region IDs Data ownership is determined based on region IDs in Listing 4.1 and 4.3. As region IDs are parameters to an inspector method, the inspector needs to compute them to pass as argument values to a method call (this is done Listing 4.1, II.3.a). The computation is based on the dimensionality of the matrix computed (DIM) and the quadrant number ($quadrant$). We extract DIM and $quadrant$ from an argument name to a call. E.g. DIM=2 (must be the same for all argument names) and $quadrant$=1 for $X01$ in $A(X01, X00, X11)$. Region IDs in the method body are computed as: $(inRegionID * 2^{DIM} + quadrant)$, where $inRegionID$ is the argument value passed to the method call. E.g. the wrapper call to the top-level inspector method $A(box, nullptr, 0, 0)$ always assigns an ID of 0 to the whole data region (Listing 4.1, III.1). The recursive call to $A(< X11 >)$ within method A in the spec is translated to a call to $A(box11, nullptr, 3, 1)$, where 3 is the new ID computed based on the incoming ID=0 ($box11$ is computed based on box, and $callStackDepth$=1 is incremented for every recursive call). The next recursive call to $A(< X11 >)$ translates to $A(box11, nullptr, 15, 2)$. As a result, the tiles are assigned IDs in Z-order [92]. Given a Z-order number, the corresponding grid coordinates can

Table 4.1.: Auxiliary methods in D2P.

Method	Description
BaseCase	Recursion terminating case. Specifies the exact function applied in computing a DP table cell. E.g. MWT computes $table_{i,j}$=min($table_{i,j}$,$table_{i,k}$+$table_{k,j}$+$weight_{i,k,j}$).
InitTable	Captures algorithm-specific initialization of the DP table. E.g. MWT initializes all the elements on the main diagonal to zero.
GetScore	Fetches the optimal score. Aggregate functions MAX, MIN are provided to fetch the maximum/minimum value among the distributed DP table cells. Specific cell content can be obtained using GetCell(i,j). In MWT, GetCell(0,N-1) fetches the optimal score, since the top-right corner cell always stores the optimal score.
ReadInput	Reads input-data. E.g. MWT reads 2D coordinates of points from a file

be efficiently computed. Based on the grid coordinates and the mapping scheme (e.g. alternating columns of tiles to different processes), tile owners can be computed.

Auxiliary methods The base case was explained earlier in section 4.2. Table 4.1 describes other auxiliary methods, which capture application-specific properties such as initializing the DP table, fetching the optimal score computed, and reading inputs. As these methods are free from the concerns of distribution, we require end-users to implement them and augment the generated code. We also provide their reference implementations for five benchmarks.

Autogen Writing a specification such as the one for the MWT problem shown in Figure 4.3 is not straightforward. Autogen is a tool that automatically discovers recursive DP algorithms for a subset of DP problems belonging to the *Fractal-DP* class [74]. Autogen takes as input an iterative code snippet consisting of only the `for` loop structure (similar to lines [3-4,7-8] in Figure 4.2), and produces a pseudocode similar to that of Figure 4.3, but with additional `parallel` annotations describing method calls that can be executed in parallel. The Autogen paper has more details on how Autogen infers recursive methods and extracts parallelism.

We design the D2P Codegen to parse Autogen's output. Thereby, we create an end-to-end system for generating distributed-memory codes for a wide variety of DP problems. As a result, an end-user can generate distributed-memory code for a real-world application by translating only the embedded loop structure representing the DP

table recurrence equation. Currently we rely on Autogen's `parallel` annotations in its output to exploit intra-task parallelism. We could extend our inter-task parallelism extraction scheme to identify parallelism within a method body in future. Note that we use Autogen primarily as a tool to generate specifications for D2P. In the next Section, we evaluate D2P with a specification that Autogen cannot generate.

4.5 Evaluation

We present the evaluation results of D2P with five DP based algorithms. These algorithms are drawn from the domains of bio-informatics, computational geometry, and computer science. The algorithms differ in their data dependency pattern and are spread over the different categories of DP problems mentioned in Galil et al. [96]. We present the scalability results and show case studies comparing D2P implementation of a benchmark with i) DPX10 [79], a generic framework for implementing distributed-memory DP algorithms and ii)a hand-written distributed-memory program [85].

Benchmarks

i **Minimum Weight Triangulation (MWT)** [93] is a triangulation algorithm, detailed in Section 4.2.

ii **Matrix Chain Multiplication (MCM)** [94] finds the optimal way to associate a sequence of matrix multiplications. Computing a single DP table cell requires reading from $\mathcal{O}(N)$ cells in both MWT and MCM, and they compute only the upper triangular matrix of the DP table.

iii **Smith-Waterman Local Alignment (SW)** [97] determines the similarity of two DNA (or amino acid) sequences. All pairs of possible *subsequences* from both the sequences are compared and scored rather than considering whole sequences. The algorithm finds local regions within the sequences having an optimal similarity score. In global sequence alignment algorithms such as Needleman-Wunsch (NW), the whole sequence is considered. The recursive formulations for both SW and

NW are the same and the implementation differs only in the terminating case. Hence, we evaluate only SW for scalability.

iv **Floyd-Warshall All Pairs Shortest Path (APSP)** [98] finds the shortest path between every pair of vertices in a graph. APSP computes the entire DP table matrix. SW and APSP are instances of the *gap* problem [96] where, SW reads $\mathcal{O}(1)$ and APSP reads $\mathcal{O}(N)$ cells to compute a cell.

v **RNA Problem (RNA)** [99] predicts the structure produced as an RNA strand folds onto itself. Our RNA implementation stores similarity values of all possible pair of subsequences to account for multi-loops, leading to a 4-dimensional DP table. Computing a DP table cell requires reading $\mathcal{O}(1)$ cells.

We use both synthetic and real world data sets [100, 101]. Among synthetic data sets, MWT uses a convex hull of randomly generated points in 2D space, and MCM uses randomly generated integer weights in the range 0 to 1000. SW uses ecoli sequences NC_000913.2E and BA000007.2, and RNA uses mRNA sequence of fruit-fly [101]. APSP uses Autonomous Systems (AS) relationships graph from the ISP data set [100].

4.5.1 Methodology

The distributed-memory (DM) implementations of DP algorithms are based on the Autogen produced shared-memory pseudocodes for all the benchmarks except MCM. In case of MCM, Autogen fails to discover a recursive algorithm since the MCM recurrence does not belong to the *Fractal-DP* class. So we wrote the specification for MCM with hints from the MWT specification, as both MWT and MCM belong to a similar category of DP problems. Our baselines are shared-memory implementations of recursive DP algorithms. Being recursive implementations, they are optimized for locality and represent the best single process sequential implementations of each benchmark among the approaches we tested. A comparison with sequential iterative

Table 4.2.: Number of tasks and the unroll depth used in strong-scaling experiments. The unroll depth used for MCM is the same as that of MWT and hence, is not shown.

Processes	2D Grid						4DGrid	
	MWT		SW		APSP		RNA	
	Unroll Depth	Tasks	Unroll Depth	Tasks	Unroll Depth	Tasks	Unroll Depth	Tasks
1	1	1	0	1	0	1	0	1
2	1	3	1	9	1	8	1	11
4	2	14	2	49	2	64	2	191
8	3	92	3	225	3	512	3	3,431
16	4	696	4	961	4	4,096	3	3,431
32	5	5,488	5	3,969	5	32,768	4	59,231
64	6	43,744	6	16,129	6	262,144	4	59,231
128	7	349,632	7	65,025	6	262,144	4	59,231
256	7	349,632	8	261,121	6	262,144	4	59,231
512	7	349,632	9	1,046,530	7	2,097,152	4	59,231

implementations for a subset of the benchmarks confirmed this: our recursive baselines (1_rec) of MCM and MWT were 1.08× and 1.36× faster than their respective iterative sequential implementations. We assume that the results hold for remaining benchmarks based on the findings in the Autogen paper.

The overall computation in a DP application can be divided into three steps: 1) table initialization, 2) table computation, and 3) backtracking. Backtracking is an important step constructing the optimal solution based on the optimal scores computed in step 2. As step 2 is computationally dominant in a majority of the DP problems, step 2 alone is timed in all performance measurements unless otherwise noted. The runtimes are measured using wall-clock time and every configuration of a test is run until a steady state is achieved.

Development and execution environment The DM implementations of all benchmarks use C++11, Intel MPI [102], and Intel CilkPlus [95]. We compile the programs with mpiicpc, a wrapper compiler for ICC 16.0.3. The experiments are run on Bridges, an XSEDE [103] cluster of over 752 nodes having Intel-OPA interconnect. We accessed up to 32 nodes of this cluster. Each node of the cluster runs CentOS Linux 7.4.1708, contains two 14-core Intel-E5-2695v3 processors with 128GB main memory.

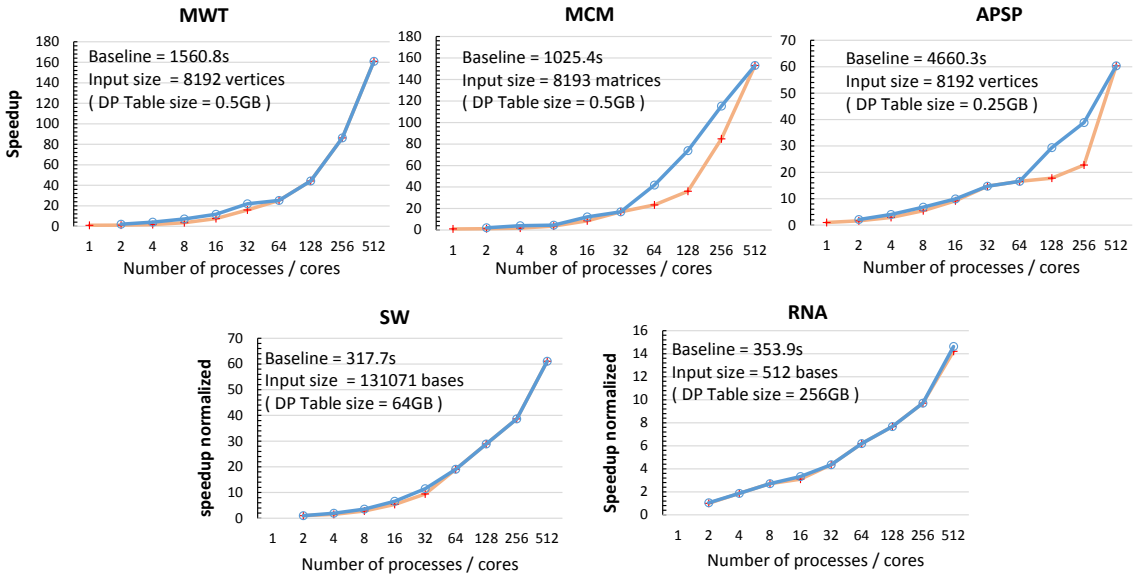


Fig. 4.5.: Strong scaling in D2P benchmarks showing speedup of single Cilk worker/process configuration over baselines. SW and RNA use baseline runtimes normalized w.r.t. 2-process (2_DM) run instead of using the 1_rec baseline runtimes. Also shown is a comparison of speedups obtained with the default unrolling depth, and the best unrolling depth empirically measured.

4.5.2 Scalability

Strong scaling In these experiments, we measure the performance of the benchmarks with increasing number of computing resources (processes, Cilk workers) for a fixed input size. The input sizes are chosen such that slowest DM executions complete in a reasonable amount of time.

Figure 4.5 shows strong-scaling with the default unrolling strategy (D based on number of processes) and 1 Cilk worker per process. The Figure also shows strong-scaling with the best value of D determined empirically for each process configuration. As single-process runs do not unroll recursion, best D is not applicable and hence, is not shown. The plots show speedup of a DM execution over the sequential recursive baseline (1_rec). In scenarios when input sizes prohibit allocating the DP table in a contiguous heap-space, we use a two-process DM run (2_DM) as the baseline.

As the results show, D2P implementations scale well given the inherently strong data dependencies in DP problems. We also observe that D2P's default unrolling strategy is sufficient. The speedup curve with default D closely follows the best D in most cases. As unrolling is commensurate with the number of processes, at larger number of processes, there is more unrolling. Deeper unrolling (in search of better D) at large number of processes causes an increase in preprocessing overhead, which outweighs the benefits obtained from increased available parallelism. As a result, best D eventually matches the default D. Since RNA uses a 4D table, with every level unrolled, there is a $16\times$ increase in the DP table tiles. So, the best and the default D are same from level 2 onwards. Hence, we do not see an observable difference in performance. Table 4.2 shows the unroll depth and number of tasks created for a given process configuration in strong-scaling experiments.

We study the effect of intra-task parallelism with multiple Cilk workers per process. Table 4.3 shows the speedups obtained with multiple Cilk workers and compares them with the best speedup numbers shown in Figure 4.5. The maximum number of cores used is the same across both single and multiple Cilk worker configurations. We

Table 4.3.: Details of (i) baseline (1_*rec*) and preprocessing overhead (*Pre*) runtimes in seconds and (ii) Comparison of speedups (×) obtained with 1 and W Cilk workers per process over baseline runtimes. The single Cilk worker per process numbers are from Figure 4.5. W represents the workers corresponding to the best run obtained from a sweep of 1 to 16 Cilk workers.

Processes	MCM			MWT			SW			APSP			RNA		
1_rec	1025.4s			1560.8s			NA			4570.6s			NA		
	Pre	Cilk-workers		Pre	Cilk-workers		Pre	Cilk-workers		Pre	Cilk-workers		Pre	Cilk-workers	
		1	W		1	W		1	W		1	W		1	W
1_DM	0.11s	0.9×	7.2×	0.1s	0.9×	4.4×	NA	NA	NA	12.4s	0.9×	7.8×	NA	NA	NA
2_DM	0.05s	1.2×	10.4×	0.04s	1.2×	7.3×	8.2s	1×	3.2×	6.1s	1.6×	17.1×	10.5s	1×	6.6×
4_DM	0.02s	1.8×	17.4×	0.02s	1.7×	14.8×	4.4s	1.5×	2.9×	3.4s	2.9×	28.9×	3.2s	1.9×	7.8×
8_DM	0.01s	3.6×	38.8×	0.01s	3.5×	33.8×	2.3s	2.8×	3.9×	1.8s	5.4×	46.8×	3.2s	2.7×	9.5×
16_DM	0.01s	8.5×	75.9×	0.01s	7.5×	75.4×	1.2s	5.3×	7.1×	0.98s	9.2×	55.3×	4.7s	3.1×	7.1×
32_DM	0.01s	16.9×	96.7×	0.02s	15.8×	63.2×	0.9s	9.4×	7.2×	0.55s	14.8×	58.8×	3.3s	4.5×	12.2×
64_DM	0.02s	23.3×	160.2×	0.2s	25.2×	108.4×	0.6s	19.1×	10.2×	0.68s	16.6×	49.4×	2.5s	6.8×	12.5×
128_DM	0.08s	36.1×	218.2×	0.28s	44.2×	289.1×	0.5s	28.9×	12.6×	0.42s	17.8×	40.4×	2s	7.8×	13.6×
256_DM	0.58s	84.8×	183.1×	0.04s	86.2×	124.8×	0.4s	38.6×	16.2×	0.27s	22.7×	30.2×	1.7s	9.9×	11.9×
512_DM	0.75s	153.1×	63.6×	0.13s	160.9×	84.4×	0.9s	61.1×	8.2×	1.8s	60.3×	23.1×	1.5s	14.2×	3.8×

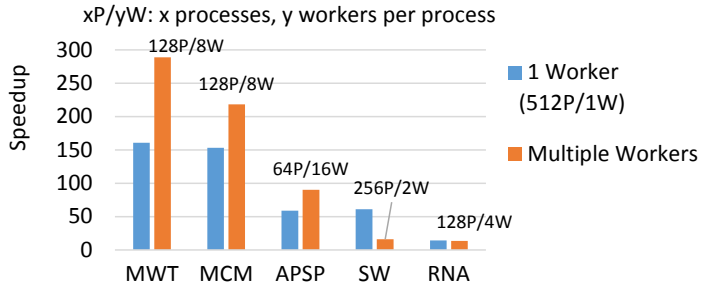


Fig. 4.6.: Summarizing Table 4.3 results. Exploiting intra-task parallelism is necessary in compute-bound benchmarks.

performed a sweep of 1 to 16 Cilk workers per process at each scale, however, we show here only the best numbers (W), which were 16 Cilk workers for up to 64 processes and 8, 4, and 2 Cilk workers for 128, 256, and 512 processes respectively. Figure 4.6 summarizes the results and shows the configurations that yield these runtimes.

Our experiments showed that allocating parallelism to additional processes rather than additional Cilk workers consistently gave better performance in some benchmarks, even though using more processes requires more communication. This is a consequence of the tradeoff between shared-memory parallelism vs. communication-computation overlap mentioned earlier (Section 4.4, intra-task parallelism). MWT, MCM, and APSP are compute-bound and hence, due to better utilization of available shared-memory parallelism, multiple Cilk workers yield almost double the speedups (280.1×, 218.2×, and 90.3× resp.) compared to single Cilk worker runs. Additional processes rather than additional Cilk workers yield better performance in memory-bound benchmarks (SW and RNA). We also measure memory contention and find that the intra-node overheads associated with communication do not play a role in memory-bound benchmarks not seeing improved speedups with multiple Cilk workers. We observed that the contention *decreased* across all benchmarks—memory- and compute-bound—, since adding more Cilk workers meant reducing the number of processes and hence, the number of communicating entities per node.

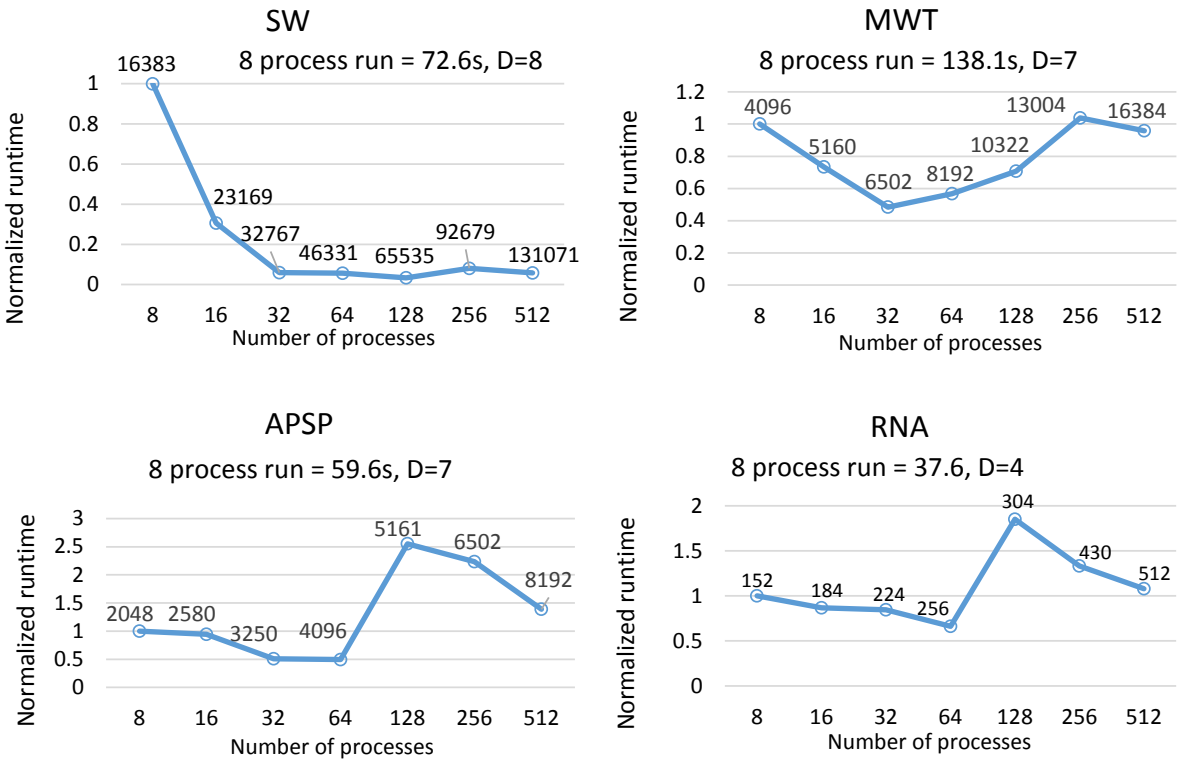


Fig. 4.7.: Weak scaling in D2P benchmarks. Y-axis shows the normalized runtime w.r.t. the 8-process run. The data-labels show the input size used for that execution.

Overall, we see a 89.6× geomean speedup in the default configuration, which further increases to 125.4× when we exploit intra-task parallelism. The addition of more Cilk workers to exploit intra-process parallelism may be necessary in order to get the absolute best performance. The results reflect D2P's use of coarse-grained, task-level parallelism at the level of leaves of the tree and Cilk workers to exploit an intra-process, sub-task parallelism available within a task.

Weak Scaling In these experiments, we increase the input size with increasing number of processes while keeping the per-process computation fixed. We also override the default D2P configuration and keep the unroll depth D (problem decomposition) constant. As a result, the runtime sometimes decreases with more processes due to more effective utilization of available parallelism. In weak-scaling experiments in

general, we expect the communication overhead to increase proportional to the number of processes. In D2P benchmarks, we have an additional source of communication overhead: as some benchmarks read $\mathcal{O}(N)$ (APSP, MWT, MCM) data to compute a cell, the amount of communication *per cell* can increase with scale. Additionally, when the input size is not a perfect power-of-two the recursive decomposition does not evenly divide the table, so partitioning tasks/sub-tasks computing these tiles can result in load-imbalance. As a result, we expect the overall runtime to increase with scaling up inputs and processes.

Figure 4.7 confirms this behavior. We see that the runtimes show different degrees of increase for different benchmarks. The 8 process run in SW uses an input sequence length of 16,384, which is perfect-power-of-two. Because SW is $\mathcal{O}(N^2)$ algorithm, the next perfect-power-of-two input size is used when the processes are scaled to 32. We consider scaling from 8-32-128-512 as an indicator of true weak-scaling since these correspond to perfect load-balance. For MWT and APSP, $\mathcal{O}(N^3)$ algorithms, 8-64-512 process runs represent the load-balanced configurations. In RNA, which is $\mathcal{O}(N^4)$, 32 and 512 process runs are each perfectly balanced.

In a perfectly load-balanced scenario, SW, MWT, APSP, and RNA weak-scale differently. SW weak-scales the best and shows a flat scaling due to a $\mathcal{O}(1)$ amount of communication per cell. Even though MWT and APSP have both $\mathcal{O}(N)$ dependencies in computing a cell, MWT shows 1.7× increase while APSP shows a 2.8× increase in runtime. This is because the communication overhead is reduced in MWT, since, it computes only the upper triangular matrix. RNA, despite computing $\mathcal{O}(N^4)$ cells, shows a modest 1.3× increase in runtime. This is because, like SW, each cell in RNA depends on $\mathcal{O}(1)$ other cells.

Preprocessing overheads and Backtracking the preprocessing time due to unrolling, dependency analysis, and partitioning remained within 1.7% of the DP table computation time in most cases. Note that all our performance numbers include this preprocessing time. We also implement backtracking manually in the D2P generated

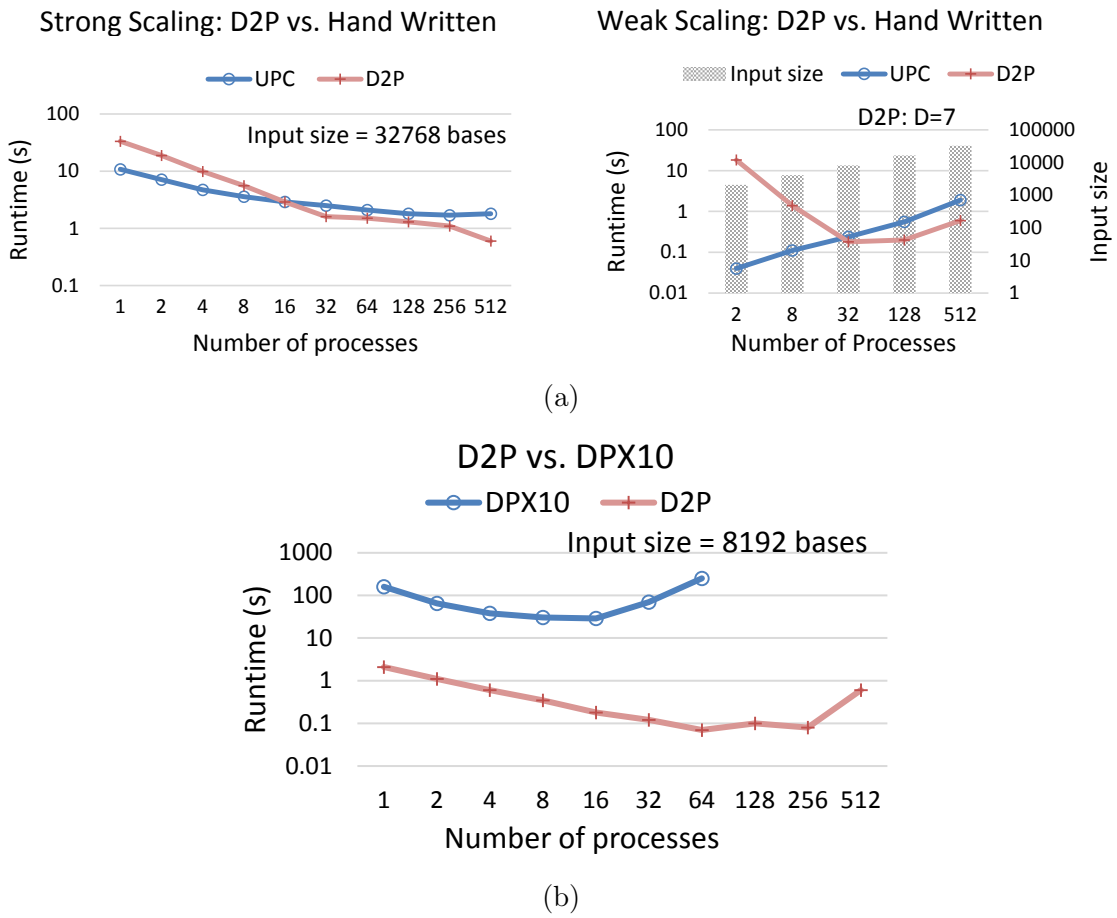


Fig. 4.8.: Case studies comparing D2P with other systems.

code and observe that for representative benchmarks (MWT, SW, and APSP), backtracking time at any scale remains within 2.1% of the DP table computation time, thus highlighting that table computation still dominates other steps.

4.5.3 Case Studies

D2P vs. Hand_Written We compare SW implementation of D2P with UPC, a highly optimized, hand-written, distributed-memory implementation of iterative SW algorithm [85]. UPC takes a static approach to task partitioning and communication. The master-slave based implementation in UPC divides the DP table (row-wise) evenly

among slaves and assigns any undivided (remaining) rows to the master. As a result, a process depends on only a single row of data from one other process owning the adjacent DP table row above. In contrast, note that a producer in D2P sends an entire tile of data to the consumer. The slaves communicate their results back to the master, who computes any assigned rows based on the results received. A pipeline-parallel scheme divides the rows further into blocks and helps exploit intra-process parallelism.

Figure 4.8a shows strong- and weak-scaling results. These measurements were obtained with 95% confidence interval with 0.08% margin of error. We chose the maximum input size permissible in UPC. Since UPC adopts a data-parallel approach, each process in UPC can only allocate a DPTable of a maximum size of 4GB in heap and this corresponds to an input size of 32768 bases. We observe that D2P scales better than UPC. Because of the default terminating case computing a single cell and the memory-bound nature of SW requiring deeper unrolling (Section 4.5.2), D2P generated code is slower at smaller number of processes. At larger number of processes, D2P outperforms UPC due to aggressive unrolling.

D2P vs. DPX10 [79] DPX10 is the best available framework for parallelizing *iterative* DP programs on distributed-memory systems. Like D2P, DPX10 exposes a set of APIs to the end-user, who specifies existing parallelism pattern and implements the iterative algorithm. Similar to D2P, DPX10 captures node-level and thread-level parallelism, however, through X10 Places and pthreads respectively.

We compare the performance of SW benchmark only, since, DPX10 currently supports only DP programs with 2-dimensional DP table and $\mathcal{O}(1)$ dependency in computing a cell (2D/0D) [79]. DPX10 takes a dynamic approach to task management through a master-slave implementation: first, a master process creates a DAG of tasks based on the parallelism pattern. The tasks are then distributed among slaves, who report back to the master with results. The master and every slave coordinate task execution with the help of an additional per-slave (and master) scheduler processes.

Figure 4.8b shows the strong scaling results. We show the best performing DPX10 run among different choices of X10 threads per place. DPX10 runs timed-out at larger scales and hence, are not shown. Here, D2P is significantly better (up to 412× speedup when best runs are compared) than DPX10 in raw performance numbers as well as scalability. We believe that the design and implementation choices (dynamic partitioning and execution of tasks, dedicated schedulers) in DPX10 affect its performance. Overall, the design choices (owner-computes, finalizing the partitioning of tasks before their execution) help D2P outperform DPX10.

4.6 Related work

Distributed-memory parallel programs and simplifying their creation has been extensively studied for many decades with a focus on iterative formulations [9, 10, 76–78, 80, 90, 104]. Recent research has shown that recursive formulations have good locality properties, readily expose parallelism and hence, can be as effective as iterative codes if implemented carefully [73, 74, 89]. In D2P we identify additional properties that, when true for recursive formulations, simplify the creation of distributed-memory parallel programs. Lifflander *et al.* [73] and Bauer *et al.* [8] (Legion) use programmer specified 'effect' annotations to identify dependences and extract parallelism. The effect annotations are explicit way of specifying inclusive properties for a recursive method. D2P's unrolling and analysis generates the required annotations automatically. In [73], different recursive method calls are spliced (interleaved) effectively to improve the cache performance of recursive implementations of stencil computations on shared-memory (SHM) systems. In Legion, the annotated code is translated and the computation is mapped to a heterogenous system. To our knowledge, no other auto-parallelization tool except Legion handles recursive codes for mapping computation to distributed-memory (DM) systems. However, as Legion targets advanced programmers, it requires significant tuning. Because of the inclusive nature of data accesses in recursive formulations, effect annotations are redundant in divide-conquer algorithms with the

inclusive property. However, we could use annotations to extend our code generation scheme to a broader set of recursive programs without the necessary properties.

D2P differs from existing general-purpose code generation frameworks in a few ways: D2P extracts parallelism (via unrolling), does explicit partitioning, and targets recursive divide-conquer formulations with inclusive and intersection properties. While D2P does static allocation of work (completed when inspector finishes executing at runtime), others do a runtime task management [8–10, 79]. Some frameworks [78, 90] rely on explicit specification of parallelism (e.g. `#pragma omp parallel for`) and implicit partitioning (implicit assignment of iterations to threads). While others [76, 77, 105] do not assume this information and work well on regular/irregular/mixed iterative codes. Sarkar *et al.* [76] automate program partitioning in a functional programming setting, and explore compile- and runtime scheduling techniques on multi-processors with different system architectures. Unlike these fully automated systems that translate a general SHM program to a DM program, D2P is a semi-automatic framework that translates a specification of a recursive divide-conquer algorithm to a distributed-memory program. Semi-automatic because the specification can be incomplete, thus requiring end-users to augment D2P produced code. However, D2P is not bound by any assumptions (e.g. about the maximum input size to produce a data-parallel code in [85, 90, 104]).

Dynamic programming [75] algorithms, with their application in a wide variety of domains, are the target of locality, parallelizing, and communication optimizations [74, 79, 81–84, 86, 88, 96, 106] on both SHM and DM systems. FastFlow [82], EasyHPS [106] and its successor DPX10 [79] specialize on iterative DP programs for DM systems. Program synthesis approaches [107, 108] to dynamic programs take a spec (e.g. a partial SHM program) with some omissions (holes) and fill it to produce a complete program. D2P, like these systems, works on a spec. However, the spec is high-level outline of a recursive algorithm and omits application-specific details. The omission is expected to be filled in the D2P produced code by the end-user. The inspector-executor

approach of Saltz *et al.* is used in D2P. We could augment the partitioning scheme in D2P with sophisticated inspector-executor based techniques [76, 77, 105, 109].

Galil *et al.* [96] categorize DP algorithms based on the dependency patterns and design SHM parallel iterative algorithms. Chowdhury *et al.* [74, 88, 89] design SHM parallel recursive DP algorithms and automate their design too. In addition, they show that SHM parallel recursive DP implementations adapt better in the presence of cache sharing, are an order of magnitude faster, and have more predictable runtimes than their tiled iterative counterparts [74]. The parallel DP formulations (both iterative and recursive) previously described perform an exact computation of the DP table. In contrast, Maleki *et al.* take an approximate-computing based approach to parallelize DP table computation on both SHM and DM systems. They use the insight that for a narrower set of DP algorithms having the linear-tropical-dynamic-programming (LTDP) property [84], where cells depend only on a previous wavefront, a cell computation need not wait for all dependences prescribed in the exact computation to be satisfied. D2P does exact computation of the DP table. As a result, it can parallelize parenthesis, APSP, and RNA problems, which have dependencies on all the previous wavefronts. Majority DM parallelizing schemes based on exact computation of SW [110, 111] exploit an application-level, coarse-grained parallelism available. They comparing a sequence against a database of sequences. The SW implementation of D2P, in contrast, compares (and parallelizes) two sequences as required by the SW algorithm.

4.7 Conclusions

In this paper, we presented D2P, a system for auto-generating distributed-memory implementations of recursive divide-conquer algorithms with inclusive and intersection properties. We showed how D2P can be used to generate those implementations from specifications. We also showed how D2P generates tasks, partitions them, and inserts communication. Finally, we evaluated D2P with recursive formulations of

DP algorithms showing that the generated implementations scale well, overheads are negligible, run significantly faster than those created manually using a similar framework, and even outperform optimized hand-written codes in most cases.

5. CONCLUSIONS

In this thesis, we presented abstractions and optimizations for implementing recursive irregular programs on distributed-memory platforms.

First, we presented SPIRIT, a framework for creating distributed implementations of spatial tree based applications. SPIRIT consists of algorithms for tree distribution and their traversal for five different types of spatial trees. The APIs in SPIRIT can be used by an average programmer to quickly create distributed implementations. Evaluation of SPIRIT implementations shows that they are efficient, thanks to a host of optimizations employed: block-scheduling improves locality, message aggregation reduces communication overheads, and pipeline-parallelism and space adaptivity expose additional parallelism, and selective replication promotes load-balance by avoiding bottlenecks in the pipeline. These optimizations result in scalable implementations that can provide enhanced performance when more space is available to replicate the data. Also, results show that SPIRIT implementations far outperform those done using generic graph processing frameworks and are competitive against customized application-specific reference software.

We then introduced Treelogy, the first categorization scheme (ontology) and a benchmark suite for tree algorithms. Treelogy includes a wide variety of tree traversal kernels and presents a useful target for developing and evaluating optimizations. The ontology helps understand where and when those optimizations can be applied. Treelogy contains nine tree algorithms spanning the ontology. Implementations of the kernels using different types of trees and on hardware platforms consisting of GPUs, shared- and distributed-memory systems are considered. Results show that most of the implementations scale well taking advantage of ontology driven optimizations, performance of two-point correlation with vptrees is better than the standard, kdtree

implementation, and a case study shows that generalizing an optimization yields substantial benefits.

Finally, we presented D2P, a state-of-the-art framework for parallelizing recursive divide-conquer algorithms on distributed-memory systems: D2P automatically generates distributed-memory implementations starting from specifications of recursive divide-conquer algorithms with inclusive and intersection properties. We evaluated D2P on recursive Dynamic Programming (DP) algorithms. D2P leverages the structural properties of recursive DP algorithms to overcome the challenges associated with distributed-memory code generation. Results show that D2P generated implementations run significantly faster than those done using similar frameworks and even outperform hand-written codes in most cases.

The research presented in this thesis has explored topics in high-performance computing, programming languages, and distributed systems. It has touched upon a number of application domains such as data mining, numerical computing, statistics, computer graphics, and bioinformatics. While the interest in graph- and tree-based traversals from various domains is increasing, we hope that i) SPIRIT and Treelogy provide insights into creating efficient tree algorithm implementations at scale and ii) D2P's implementations help in efficiently solving some of the most compute- and memory-intensive problems in emerging application domains.

REFERENCES

REFERENCES

[1] M. V. Kulkarni, "The galois system: optimistic parallelization of irregular programs," Ph.D. dissertation, Cornell University, 2008.

[2] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[3] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[4] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30.

[6] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[7] "http://www.boost.doc/org/libs."

[8] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–11.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[10] C. Huang and L. Kale, "Charisma: Orchestrating migratable parallel objects," in *Proceedings of the 16th international symposium on High performance distributed computing*. ACM, 2007, pp. 75–84.

[11] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, September 1975. [Online]. Available: http://doi.acm.org/10.1145/361002.361007

[12] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA*, vol. 93, no. 194, 1993, pp. 311–321.

[13] D. J. Meagher, *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*. Electrical and Systems Engineering Department Rensseiaer Polytechnic Institute Image Processing Laboratory, 1980.

[14] O. STEPHEN, M., "Five balltree construction algorithms," International Computer Science Institute, Tech. Rep. 89-063, November 1989.

[15] N. Hegde, J. Liu, and M. Kulkarni, "SPIRIT: A Framework for Creating Distributed Recursive Tree Applications," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: ACM, 2017, pp. 3:1–3:11. [Online]. Available: http://doi.acm.org/10.1145/3079079.3079095

[16] N. Hegde, J. Liu, K. Sundararajah, and M. Kulkarni, "Treelogy: A benchmark suite for tree traversals," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 227–238.

[17] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Grappa: A latency-tolerant runtime for large-scale irregular applications," in *International Workshop on Rack-Scale Computing (WRSC w/EuroSys)*, 2014.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[19] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 463–482.

[20] A. Fidel, N. M. Amato, L. Rauchwerger *et al.*, "The stapl parallel graph library," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 46–60.

[21] J. Barnes and P. Hut, "A hierarchical o (n log n) force-calculation algorithm," *Nature*, 1986.

[22] B. Walter, K. Bala, M. Kulkarni, and K. Pingali, "Fast agglomerative clustering for rendering," in *IEEE Symposium on Interactive Ray Tracing (RT)*, August 2008, pp. 81–86.

[23] A. G. Gray and A. Moore, "N-body problems in statistical learning," in *Advances in Neural Information Processing Systems (NIPS) 13*, December 2001, pp. 521–527.

[24] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '05, 2005, pp. 15–22. [Online]. Available: http://doi.acm.org/10.1145/1071866.1071869

[25] M. Lichman, "UCI machine learning repository," 2013.

[26] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. Quinn, "Massively parallel cosmological simulations with changa," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[27] T. Quinn, Personal correspondence, 2016.

[28] M. D. Dikaiakos and J. Stadel, "A performance study of cosmological simulations on message-passing and shared-memory multiprocessors," in *Proceedings of the 10th international conference on Supercomputing*. ACM, 1996, pp. 94–101.

[29] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 456–471.

[30] S. Gupta and V. K. Nandivada, "IMSuite: A benchmark suite for simulating distributed algorithms," *Journal of Parallel and Distributed Computing*, vol. 75, no. 0, pp. 1 – 19, Jan. 2015.

[31] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Am++: A generalized active message framework," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 401–410.

[32] J. Chhugani, C. Kim, H. Shukla, J. Park, P. Dubey, J. Shalf, and H. D. Simon, "Billion-particle simd-friendly two-point correlation on large-scale hpc cluster systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 1.

[33] J. Pantaleoni, L. Fascione, M. Hill, and T. Aila, "Pantaray: fast ray-traced occlusion caching of massive scenes," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, p. 37, 2010.

[34] M. Amor, F. Argüello, J. López, O. Plata, and E. L. Zapata, "A data parallel formulation of the barnes-hut method for n-body simulations," in *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*. Springer, 2001, pp. 342–349.

[35] M. Aly, M. Munich, and P. Perona, "Distributed kd-trees for retrieval from very large image collections," in *Proceedings of the British Machine Vision Conference (BMVC)*, 2011.

[36] M. S. Warren and J. K. Salmon, "Astrophysical n-body simulations using hierarchical tree data structures," in *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1992, pp. 570–576.

[37] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity," *Journal of Parallel and Distributed Computing*, vol. 27, no. 2, pp. 118–141, 1995.

[38] J. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm," *nature*, vol. 324, p. 4, 1986.

[39] V. Rokhlin, "Rapid solution of integral equations of classical potential theory," *Journal of Computational Physics*, vol. 60, no. 2, pp. 187–207, 1985.

[40] A. G. Gray and A. Moore, "N-body problems in statistical learning," in *Advances in Neural Information Processing Systems (NIPS) 13*, December 2001, pp. 521–527.

[41] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *ACM Sigmod Record*, vol. 29, no. 2. ACM, 2000, pp. 1–12.

[42] K. Alsabti, S. Ranka, and V. Singh, "An efficient k-means clustering algorithm," 1997.

[43] K. Toru, L. Gunho, A. Hiroki, A. Setsuo, and P. Kunsoo, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, A. Amir, Ed. Springer-Verlag London, UK, 07 2001, pp. 181–192.

[44] X. Zhang and A. A. Chien, "Dynamic pointer alignment: Tiling and communication optimizations for parallel pointer-based computations," in *ACM SIGPLAN Notices*, vol. 32, no. 7. ACM, 1997, pp. 37–47.

[45] N. Kumar, L. Zhang, and S. Nayar, "What is a good nearest neighbors algorithm for finding similar patches in images?" in *Computer Vision–ECCV 2008*. Springer, 2008, pp. 364–378.

[46] T. Hamada, K. Nitadori, K. Benkrid, Y. Ohno, G. Morimoto, T. Masada, Y. Shibata, K. Oguri, and M. Taiji, "A novel multiple-walk parallel algorithm for the barnes–hut treecode on gpus – towards cost effective, high performance n-body simulation," *Computer Science - Research and Development*, vol. 24, no. 1, pp. 21–31, 2009. [Online]. Available: http://dx.doi.org/10.1007/s00450-009-0089-1

[47] Y. Jo and M. Kulkarni, "Automatically enhancing locality for tree traversals with traversal splicing," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 355–374. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384643

[48] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for gpu execution of tree traversals," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503223

[49] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH computer architecture news*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

[50] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[51] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 65–76.

[52] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User?s Group (CUG)*, 2010.

[53] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on.* IEEE, 2012, pp. 141–151.

[54] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on.* IEEE, 2015, pp. 44–55.

[55] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, "The tao of parallelism in algorithms," in *ACM Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 12–25.

[56] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey, "Cache-conscious frequent pattern mining on modern and emerging processors," *The VLDB Journal*, vol. 16, no. 1, pp. 77–96, 2007.

[57] H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On visible surface generation by a priori tree structures," in *ACM Siggraph Computer Graphics*, vol. 14, no. 3. ACM, 1980, pp. 124–133.

[58] P. Bieganski, J. Riedl, J. V. Cartis, and E. F. Retzel, "Generalized suffix trees for biological sequence data: applications and implementation," in *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, vol. 5, Jan 1994, pp. 35–44.

[59] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 463–482. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048104

[60] N. Bhatia and Vandana, "Survey of nearest neighbor techniques," in *International Journal of Computer Science and Information Security*, ser. IJCSIS, vol. 8, 2010, pp. 302–305.

[61] L. Ying, "A pedestrian introduction to fast multipole methods," *Science China Mathematics*, vol. 55, no. 5, pp. 1043–1051, 2012.

[62] J. Liu, N. Hegde, and M. Kulkarni, "Hybrid cpu-gpu scheduling and execution of tree traversals," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2925426.2926261

[63] J. Makino, "Vectorization of a treecode," *Journal of Computational Physics*, vol. 87, no. 1, pp. 148–160, 1990.

[64] M. Höhl, S. Kurtz, and E. Ohlebusch, "Efficient multiple genome alignment," *Bioinformatics*, vol. 18, no. suppl 1, pp. S312–S320, 2002.

[65] C. Borgelt, "Frequent item set mining," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 2, no. 6, pp. 437–456, 2012.

[66] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. ACM, 2003, pp. 245–257.

[67] N. Hegde, J. Liu, and M. Kulkarni, "Spirit: A runtime system for distributed irregular tree applications," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 51:1–51:2. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851177

[68] K. Sundararajah, L. Sakka, and M. Kulkarni, "Locality transformations for nested recursive iteration spaces," in *Proceedings of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://dx.doi.org/10.1145/3037697.3037720

[69] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[70] S.-H. Lim, S. Lee, G. Ganesh, T. C. Brown, and S. R. Sukumar, "Graph processing platforms at scale: Practices and experiences," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 42–51.

[71] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on gpu with bvh-based packet traversal," in *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, ser. RT '07, 2007, pp. 113–118. [Online]. Available: http://dx.doi.org/10.1109/RT.2007.4342598

[72] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based barnes hut n-body algorithm," in *GPU Computing Gems Emerald Edition*. Elsevier Inc., 2011, pp. 75–92.

[73] J. Lifflander and S. Krishnamoorthy, "Cache locality optimization for recursive programs," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 1–16.

[74] R. Chowdhury, P. Ganapathi, J. J. Tithi, C. Bachmeier, B. C. Kuszmaul, C. E. Leiserson, A. Solar-Lezama, and Y. Tang, "Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, p. 10.

[75] R. Bellman, *Dynamic Programming*, 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[76] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989.

[77] M. Ravishankar, R. Dathathri, V. Elango, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Distributed memory code generation for mixed irregular/regular computations," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 65–75.

[78] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006, pp. 119–128.

[79] C. Wang, C. Yu, S. Tang, J. Xiao, J. Sun, and X. Meng, "A general and fast distributed system for large-scale dynamic programming applications," *Parallel Computing*, vol. 60, pp. 1–21, 2016.

[80] U. Bondhugula, "Automatic Distributed Memory Code Generation using the Polyhedral Framework," Indian Institute of Science, Bangalore, Tech. Rep. IISc-CSA-TR-2011-3, 2011.

[81] W. Zhou and D. K. Lowenthal, "A parallel, out-of-core algorithm for RNA secondary structure prediction," in *Parallel Processing, 2006. ICPP 2006. International Conference on*. IEEE, 2006, pp. 74–81.

[82] M. Aldinucci, M. Meneghin, and M. Torquati, "Efficient Smith-Waterman on multi-core with FastFlow," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 2010, pp. 195–199.

[83] M. Baker, A. Welch, and M. G. Venkata, "Parallelizing the smith-waterman algorithm using openshmem and mpi-3 one-sided interfaces," in *Workshop on OpenSHMEM and Related Technologies*. Springer, 2014, pp. 178–191.

[84] S. Maleki, M. Musuvathi, and T. Mytkowicz, "Parallelizing dynamic programming through rank convergence," *ACM SIGPLAN Notices*, vol. 49, no. 8, pp. 219–232, 2014.

[85] "SmithWaterman with OpenMPI and OpenMP," https://www.alexjf.net/projects/distributed-systems/smith-waterman-openmp-and-openmpi/.

[86] K. Hamidouche, F. M. Mendonca, J. Falcou, A. C. M. A. de Melo, and D. Etiemble, "Parallel Smith-Waterman Comparison on Multicore and Manycore Computing Platforms with BSP++," *International Journal of Parallel Programming*, vol. 41, no. 1, pp. 111–136, Feb 2013. [Online]. Available: https://doi.org/10.1007/s10766-012-0209-6

[87] M. Ziv-Ukelson, I. Gat-Viks, Y. Wexler, and R. Shamir, "A faster algorithm for rna co-folding," in *International Workshop on Algorithms in Bioinformatics*. Springer, 2008, pp. 174–185.

[88] R. A. Chowdhury and V. Ramachandran, "Cache-efficient dynamic programming algorithms for multicores," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 2008, pp. 207–216.

[89] R. A. Chowdhury, H.-S. Le, and V. Ramachandran, "Cache-oblivious dynamic programming for bioinformatics," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 7, no. 3, pp. 495–510, 2010.

[90] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff, "A hybrid approach of OpenMP for clusters," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 75–84.

[91] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *ACM Sigplan Notices*, vol. 33, no. 5. ACM, 1998, pp. 212–223.

[92] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," 1966.

[93] G. Klincsek, "Minimal triangulations of polygonal domains," *Annals of Discrete Mathematics*, vol. 9, pp. 121–123, 1980.

[94] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[95] "Intel Cilk Plus," `https://www.cilkplus.org`.

[96] Z. Galil and K. Park, "Parallel algorithms for dynamic programming recurrences with more than O (1) dependency," *Journal of Parallel and Distributed Computing*, vol. 21, no. 2, pp. 213–222, 1994.

[97] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283681900875

[98] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[99] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman, "Algorithms for loop matchings," *SIAM Journal on Applied mathematics*, vol. 35, no. 1, pp. 68–82, 1978.

[100] "The CAIDA AS Relationships Dataset, Nov'5, 2007," http://www.caida.org/data/as-relationships/.

[101] S. T. Sherry, M.-H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin, "dbSNP: the NCBI database of genetic variation," *Nucleic acids research*, vol. 29, no. 1, pp. 308–311, 2001.

[102] "Intel MPI Libraries:," https://software.intel.com/en-us/intel-mpi-library.

[103] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "XSEDE: Accelerating Scientific Discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, Sept.-Oct. 2014. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MCSE.2014.80

[104] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with mpi," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 712–725.

[105] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky, "An approach for code generation in the sparse polyhedral framework," *Parallel Computing*, vol. 53, no. C, pp. 32–57, April 2016.

[106] J. Du, C. Yu, J. Sun, C. Sun, S. Tang, and Y. Yin, "EasyHPS: A multilevel hybrid parallel system for dynamic programming," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 630–639.

[107] S. Itzhaky, R. Singh, A. Solar-Lezama, K. Yessenov, Y. Lu, C. Leiserson, and R. Chowdhury, "Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016, pp. 145–164.

[108] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *ACM Sigplan Notices*, vol. 45, no. 1. ACM, 2010, pp. 313–326.

[109] K. S. Gatlin and L. Carter, "Architecture-cognizant divide and conquer algorithms," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM, 1999, p. 25.

[110] M. Noorian, H. Pooshfam, Z. Noorian, and R. Abdullah, "Performance enhancement of smith-waterman algorithm using hybrid model: Comparing the mpi and hybrid programming paradigm on smp clusters," in *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*. IEEE, 2009, pp. 492–497.

[111] A. E. Darling, L. Carey, and W. C. Feng, "The design, implementation, and evaluation of mpiblast," Los Alamos National Laboratory, Tech. Rep., 2003.

VITA

VITA

Nikhil D. Hegde

Nikhil Hegde is a Ph.D. candidate in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, USA and is advised by Prof. Milind Kulkarni. He obtained his master's degree from IIT Madras in 2005, and bachelor's degree from B.M.S. College of Engineering, Bangalore in 2002, both in Computer Science and Engineering. He also has over eight years of professional experience working in the industry developing hardware, middleware, and software for mobile platforms. He worked for Intel, Nokia, ST Microelectronics, and AdsFLO, a start-up that delivered mobile advertising solutions. He is interested in the broad areas of systems research including parallel and distributed computing, compilers, and programming languages.